LINE: Queueing Analysis Algorithms

Solver Internals: A Manual for Researchers

Last revision: July 11, 2026

Contents

Chapter 1

Introduction

1.1 Purpose of this manual

LINE is an open-source software package to analyze queueing models via analytical methods and simulation [22]. The user manuals of the tool (`LINE-matlab.pdf`, `LINE-java.pdf`, `LINE-python.pdf`) document the modeling language, the solver invocation interfaces, and the model features supported by each solver. They deliberately treat each solver as a black box: the user selects a solver and a method, and the tool returns performance metrics.

This manual takes the complementary viewpoint. It is oriented to researchers who wish to understand, evaluate, extend, or cite the solution algorithms implemented inside LINE. Each chapter maps one-to-one to a solver and presents:

- the solution paradigm the solver encodes, with its theoretical foundations;
- the internal software architecture, i.e., the solver class, its analyzers, its method handlers, and the numerical algorithms in the API layer that they invoke;
- the architectural flow of a solution request, illustrated by UML sequence diagrams;
- the concrete algorithms behind each value of `options.method`, with references to the scientific literature in which they were introduced or analyzed.

Readers are assumed to be familiar with queueing network theory at the level of standard textbooks [12, 73, 77] and with the modeling abstractions of LINE as described in the user manuals. Whenever this manual mentions user-facing behavior, such as option names or supported model features, the user manuals remain the authoritative reference; conversely, for the internal behavior of the algorithms, this manual supersedes the brief descriptions given there.

1.2 Relation to the existing manuals

The user manuals are organized by modeling concern: network specification, solver invocation, and output interpretation. This manual is organized by solver, mirroring the source tree. The two documents share

terminology:

- A *solver* encodes a general solution paradigm, e.g., mean-value analysis or continuous-time Markov chain analysis. Solvers are exposed to users as classes named `Solver[Name]`.
- An *analyzer* is an internal component that transforms the model representation and dispatches it to a concrete solution method. Analyzers are selected automatically based on the model class (e.g., a network containing a cache node routes to a cache analyzer).
- A *method* is a concrete algorithm selected via `options.method`, e.g., `exact`, `amva`, or `comom`. Methods are implemented in *handler* classes and often delegate the numerical core to *API algorithms*.
- The *API layer* collects reusable numerical functions organized by domain: `PFQN` (product-form queueing networks), `NPFQN` (non-product-form approximations), `MC` (Markov chains), `MAM` (matrix-analytic methods), `MAP` (Markovian arrival processes), `CACHE`, `QSYS` (single queueing systems), `POLLING`, `SN` (network structure transformations), and others.

LINE is a multi-language project. The canonical implementation is the Java JAR (`jar/` subfolder), which is also the backend called by the MATLAB and Python wrappers; MATLAB (`matlab/`) is the reference implementation used as ground truth in parity testing; a native Python implementation (`python/`) mirrors both without any JVM dependency. All implementations follow the same architecture and naming conventions, e.g., the MATLAB function `solver_mva_analyzer.m` corresponds to the Java class `Solver_mva_analyzer` and to the Python module `solver_mva_analyzer.py`. Throughout this manual we use the Java class names as canonical identifiers; the mapping to the other codebases is mechanical.

1.3 The common solution pipeline

All network solvers share the same solution pipeline, inherited from the abstract class `NetworkSolver`. A user constructs a `Network` model, instantiates a solver over it, and requests metrics, typically through `getAvgTable` or one of the finer-grained handles (`getAvg`, `getAvgQLen`, `getAvgRespT`, `getTranAvg`, and so forth). The request triggers the following steps:

1. *Structural compilation.* The object-oriented model is compiled by `Network.getStruct()` into a `NetworkStruct` instance, denoted `sn` throughout this manual. This is a flat, matrix-oriented description of the extended multiclass queueing network: stations, classes, chains, routing tables, phase-type service representations $(\boldsymbol{D}_0, \boldsymbol{D}_1)$, scheduling identifiers, capacities, and the auxiliary parameter blocks documented in the user manuals. All analyzers operate exclusively on `sn`, never on the object model; this is the model-to-model transformation boundary that decouples specification from solution.
2. *Feature admission.* The solver checks the model features present in `sn` against its feature set (`getFeatureSet`); unsupported models are rejected before any computation.
3. *Analyzer dispatch.* The solver method `runAnalyzer()` selects an analyzer based on model characteristics: the presence of cache nodes, polling stations, load dependence, an isolated open queue, or bound-computation requests each route to a specialized analyzer, otherwise the default analyzer runs.

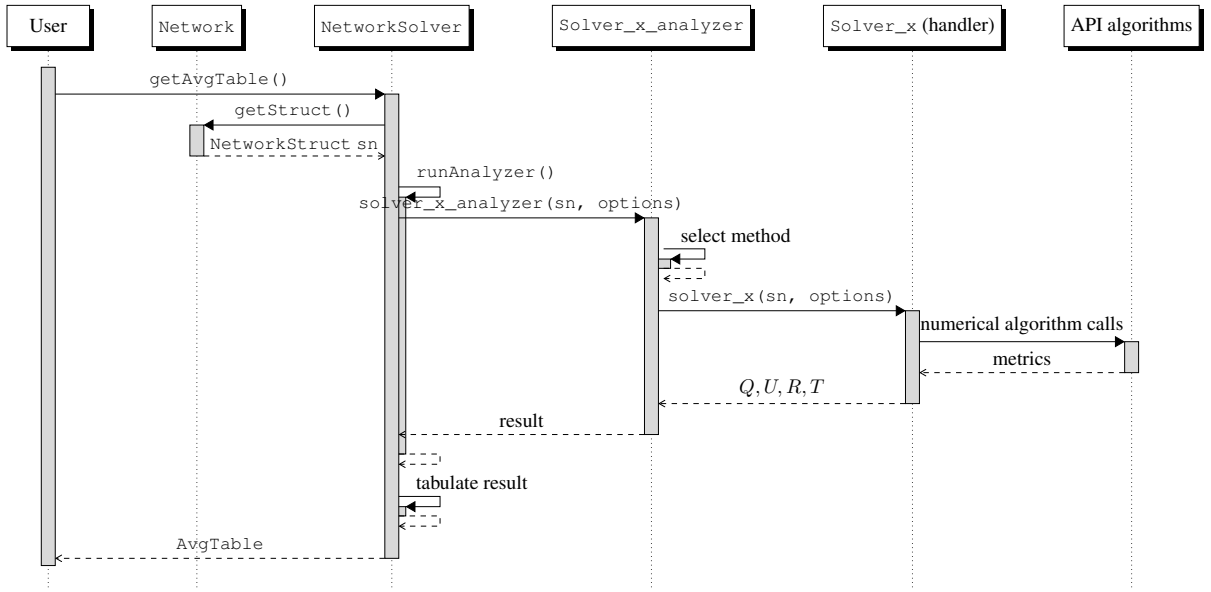

Figure 1.1: Generic solution pipeline shared by all LINE network solvers. The placeholder `x` stands for the solver name (`mva`, `nc`, `ctmc`, ...).

4. *Method dispatch.* The analyzer normalizes `options.method`, applies default-selection policies, and invokes the handler implementing the chosen algorithm.

5. *Numerical solution.* The handler evaluates the model by calls into the API layer and returns station-class matrices of mean queue lengths Q_{ir}, utilizations U_{ir}, response times R_{ir}, throughputs T_{ir}, arrival rates, and residence times.

6. *Tabulation.* The solver stores the result and materializes it into the requested table object (e.g., `NetworkAvgTable`), applying chain aggregation and visit scaling as needed.

Figure 1.1 depicts this pipeline as a UML sequence diagram; every chapter of this manual contains an analogous diagram specialized to the internal components of the corresponding solver.

Meta-solvers (`AUTO`, `ENV`, `LN`, `UQ`) deviate from this pipeline in that they do not evaluate `sn` directly: they construct an ensemble of `Network` submodels and coordinate other solvers over it, typically through the abstract class `EnsembleSolver`. LINE also includes wrappers around external tools (`JMT` [8], `LQNS` [45], `QNS`), which replace steps 4-5 with a marshalling stage that exports `sn` to the input format of the external tool, executes it as a subprocess, and parses the results back. Since their solution algorithms are external to LINE, Part III documents only the wrapper architecture, i.e., the model marshalling, the method/feature mapping, and the result parsing; the external algorithms themselves remain documented in the user manuals and in the references above.

1.4 Notation

We use the index conventions of the `NetworkStruct` reference in the user manuals: i, j range over stations (M stations), r, s over classes (R classes), c over chains, and k over phases. The population vector of a closed model is $\boldsymbol{N} = (N_1, \ldots, N_R)$ with total population N; service rates are μ_{ir}, mean service demands $D_{ir} = V_{ir}/\mu_{ir}$ where V_{ir} is the visit ratio; c_i denotes the number of servers of station i. Phase-type and Markovian arrival processes are written in $(\boldsymbol{D}_0, \boldsymbol{D}_1)$ notation [17, 91]. For a CTMC we write $\boldsymbol{Q}$ for the infinitesimal generator and $\boldsymbol{\pi}$ for its stationary probability vector, $\boldsymbol{\pi Q} = \boldsymbol{0}$, $\boldsymbol{\pi 1} = 1$.

Mean performance metrics follow the tool-wide convention: `QLen` Q_{ir} (mean number of class-r jobs at station i), `Util` U_{ir} (utilization, in $[0, 1]$ for single-server stations and reported as mean busy servers otherwise), `RespT` R_{ir} (response time per visit at the station), `ResidT` (residence time scaled by visits relative to the reference station), `ArvR` (arrival rate), and `Tput` T_{ir} (departure rate). These satisfy Little's law $Q_{ir} = T_{ir} R_{ir}$ at each station [82] and the utilization law $U_{ir} = T_{ir} D_{ir}/c_i$.

1.5 Organization

Part I covers the native solvers: `MVA` (Chapter 2), `NC` (Chapter 3), `CTMC` (Chapter 4), `SSA` (Chapter 5), `FLD` (Chapter 6), `MAM` (Chapter 7), and `LDES` (Chapter 8). Part II covers the meta-solvers `AUTO` (Chapter 9), `ENV` (Chapter 10), `LN` (Chapter 11), and `UQ` (Chapter 12). Part III covers the wrappers around external tools: `JMT` (Chapter 13), `LQNS` (Chapter 14), and `QNS` (Chapter 15).

Each chapter closes with bibliographic notes that position the implemented algorithms in the literature. To cite the LINE solver itself, we recommend referencing [22].

Part I

Native solvers

Chapter 2

SolverMVA

2.1 Overview

`SolverMVA` encodes the mean-value analysis paradigm for product-form queueing networks and its approximate extensions to non-product-form models. Exact mean-value analysis (MVA) computes mean performance metrics of closed product-form networks by a recursion over populations without ever evaluating the normalizing constant [98,99]. Approximate mean-value analysis (AMVA) replaces the population recursion with a fixed-point iteration over interpolated queue lengths, trading exactness for a cost that is independent of the population size [5,30,107]. On top of these two pillars, the solver hosts a family of specialized analyzers for bounds, caches, polling systems, isolated queueing systems, load-dependent stations, and open-network decomposition.

The solver resides in `jline/solvers/mva/` with the analyzer classes in `analyzers/` and the method handlers in `handlers/`. Its numerical core is the `PFQN` API (`jline/api/pfqn/`), complemented by `NPFQN`, `QSYS`, `CACHE`, and `POLLING` algorithms.

2.2 Software architecture

Table 2.1 lists the main classes. The entry point `SolverMVA.runAnalyzer()` inspects `sn` and routes the request to one of eight analyzers; each analyzer selects a handler according to `options.method`.

2.3 Default method selection

When `options.method='default'`, `Solver_mva_analyzer` applies the following policy, in order:

1. if the model is a single-chain closed network with blocking-after-service (BAS) drop rules, run the `sqd` handler;

2. if the model mixes open and closed classes, has only single-server stations, satisfies the BCMP product-form conditions [6], and the closed populations are integral, run exact mixed MVA

Component	Class	Role
Solver	`SolverMVA`	feature admission, analyzer dispatch, result handles
Options	`MVAOptions`	method, tolerance `iter_tol`, iteration cap `iter_max`
Result	`MVAResult`, `ProbabilityResult`	metric matrices, iteration counts, logNormConst
Analyzer	`Solver_mva_analyzer`	default dispatch for queueing networks
Analyzer	`Solver_mvald_analyzer`	load-dependent networks
Analyzer	`Solver_mva_bound_analyzer`	non-iterative bound methods
Analyzer	`Solver_mva_qsys_analyzer`	isolated open queueing systems
Analyzer	`Solver_mva_cache_analyzer`	standalone caches
Analyzer	`Solver_mva_cacheqn_analyzer`	cache-queueing networks
Analyzer	`Solver_mva_retrieval_analyzer`	delayed-hit retrieval caches
Analyzer	`Solver_mva_polling_analyzer`	polling stations
Handler	`MVARunner`	orchestration, fork-join transformations
Handler	`Solver_mva`, `Solver_mvald`	exact MVA, exact load-dependent MVA
Handler	`Solver_amva`, `Solver_amvald`	AMVA fixed-point schemes
Handler	`Solver_qna`	open-network decomposition (QNA)
Handler	`Solver_sqd`	blocking-after-service single-chain models
Handler	`Solver_mva_lcfsqn`	LCFS-PR closed networks

Table 2.1: Main classes of `SolverMVA`.

(`pfqn_mvamx`) [16], since the AMVA linearizer is known to inflate closed-class response times by double-counting open-class interference;

3. if the model is product-form with at most 4 chains and at most 20 jobs, run exact MVA;
4. otherwise, run AMVA, whose internal default is the linearizer family described below.

This policy reflects the general design rule of LINE that exact algorithms are preferred whenever their combinatorial cost is provably small, and asymptotically scalable approximations are used otherwise.

2.4 Exact MVA methods

The `exact` (alias `mva`) method evaluates closed product-form networks using the arrival theorem [99, 108]: an arriving class-r job observes the network in equilibrium with population $\boldsymbol{N} - \mathbf{1}_r$. For single-server FCFS/PS/LCFS-PR stations and infinite-server delays, the recursion is

$$R_{ir}(\boldsymbol{N}) = D_{ir}\,[1 + Q_i(\boldsymbol{N} - \mathbf{1}_r)] \quad \text{(queueing stations)}, \qquad R_{ir}(\boldsymbol{N}) = D_{ir} \quad \text{(delay stations)},$$
$$T_r(\boldsymbol{N}) = \frac{N_r}{Z_r + \sum_i R_{ir}(\boldsymbol{N})}, \qquad Q_{ir}(\boldsymbol{N}) = T_r(\boldsymbol{N})\, R_{ir}(\boldsymbol{N}), \tag{2.1}$$

iterated over all population vectors $\mathbf{0} \le \boldsymbol{n} \le \boldsymbol{N}$ in lexicographic order. The implementation is `pfqn_mva`; its time complexity is $O(MR\prod_r(N_r+1))$ and its space complexity is reduced by retaining only the layers of the population lattice needed by the recursion. Multiserver stations are handled by `pfqn_mvams`, which augments (2.1) with the exact marginal-probability recursion of queue-dependent MVA [98]; mixed open-closed models are solved by `pfqn_mvamx`, which first computes the open-class utilizations and then inflates closed-class demands by $(1 - U_i^{\text{open}})^{-1}$ following the exact BCMP decomposition [6, 16].

Exact MVA for load-dependent stations is provided by the `Solver_mvald` handler through `pfqn_mvald`, which propagates the full marginal queue-length distributions $\pi_i(n \mid \boldsymbol{N})$ across populations.

Since the load-dependent recursion is numerically unstable in the presence of near-zero marginal probabilities [24, 98], the handler monitors the probability mass and rescales it, flagging results where stabilization was applied.

2.5 AMVA methods

The `Solver_amva` and `Solver_amvald` handlers implement the approximate MVA family. All methods share the fixed-point skeleton: guess queue lengths Q_{ir}, estimate the arrival-instant queue lengths A_{ir} seen by a class-r arrival at station i, evaluate response times, apply Little's law, and repeat until the queue lengths change by less than `iter_tol`. The methods differ in the interpolation used for A_{ir}:

`bs` Bard-Schweitzer proportional estimator $A_{ir} = Q_i(\boldsymbol{N}) - \frac{Q_{ir}(\boldsymbol{N})}{N_r}$ [5, 107]. Cost $O(MR)$ per iteration; existence and uniqueness of the fixed point are known for single-class models.

`lin` The Linearizer of Chandy and Neuse [30], a three-level scheme that estimates the fractional deviations δ_{irs} between populations $\boldsymbol{N}$ and $\boldsymbol{N} - \mathbf{1}_s$ and is typically accurate to a fraction of a percent. Multiserver stations use the extensions of Conway [34] and the multiserver AMVA corrections of Suri et al. [112] (`pfqn_linearizerms`, `pfqn_conwayms`); mixed models use `pfqn_linearizermx`. The general-form variants `gflin` and `egflin` (`pfqn_gflinearizer`, `pfqn_egflinearizer`) parameterize the deviation update in the spirit of the unified framework of Cremonesi, Schweitzer and Serazzi [38], which subsumes Linearizer variants under a common template; `fli` is the fraction-line one-step variant from the same framework.

`aql` The aggregated-queue-length method of Zahorjan, Eager and Sweillam [122], which iterates on aggregate station populations.

`qd`, `qdlin`, `qli` Queue-dependent AMVA (QD-AMVA) [26]: the arrival estimator is made a function of the local queue length to interpolate load-dependent rates, yielding methods that handle multiserver and limited load-dependent stations within the fixed-point scheme (`pfqn_qd`, `qdlin` combines this with Linearizer deviations, `qli` is the queue-line variant).

`sqni` A single-queue network interpolation built on the QRF optimization view of interpolated networks [23] (`pfqn_sqni`).

`schmidt`, `schmidt-ext` The exact-arrival-theorem AMVA of Schmidt for networks with FCFS multiserver stations (`pfqn_schmidt_amva`), including its extended variant.

`ab` The Akyildiz-Bolch load-dependent approximation [2] (`pfqn_ab_amva`).

Priority classes are handled inside `Solver_amvald` by shadow-server reductions in the tradition of Sevcik [108] and the AMVA priority approximation of Eager and Lipscomb [42]. First-come first-served stations with non-exponential service use the interpolations of `pfqn_bsfcfs`, which blend the product-form response time with a c_{ir}^2-weighted correction; comparative background on such heuristics is given in [13, 117].

Load-dependent AMVA (`Solver_amvald`) supports limited load dependence (LLD), class dependence (CD), and limited joint dependence, following the factorization approach for load-dependent normalizing constants of [24]. The handler linearizes the scaling tables in `sn` (`lldscaling`, `cdscaling`, `ljcdscaling`) and embeds them in the arrival estimator.

2.6 Bound methods

`Solver_mva_bound_analyzer` serves the non-iterative bound methods, each in `.upper`/`.lower` pairs: `aba` (asymptotic bound analysis) and `bjb` (balanced job bounds) [77], `pb` (proportional bounds), `gb` (geometric bounds, a noniterative technique with error guarantees on throughput [25], algorithms `pfqn_qzgblow`/`pfqn_qzgbup`), `sb` (scaled bounds), and `harel` (the convexity-based bounds of Harel, Namn and Sturm [55], algorithm `Pfqn_harel_bounds`). These methods require no iteration and are mainly intended for optimization loops, where monotonicity and guaranteed sidedness matter more than accuracy [21].

2.7 Specialized analyzers

Queueing systems. When the model consists of a single open queue (Source, Queue, Sink), `Solver_mva_qsys_analyzer` bypasses network iteration and evaluates closed-form or quadrature results from the `QSYS` API: M/M/1, M/M/k, M/G/1 via the Pollaczek-Khinchine formula, G/M/1, and G/G/1 through a portfolio of approximations selectable by method name, including Allen-Cunneen, Kraemer and Langenbach-Belz [72], Kobayashi's diffusion approximation [68], Kingman bounds, Marchal, Heyman, Gelenbe, Kimura, and Myskja variants, plus Cosmetatos and Whitt corrections for G/G/k. Scheduling-sensitive metrics for M/G/1 (SRPT, FB, priorities) follow the exact transform analyses surveyed in [93, 121]; the SRPT mean response time is computed by the Schrage-Miller quadrature [106].

Polling. `Solver_mva_polling_analyzer` evaluates exhaustive, gated, and decrementing (limited) polling stations with switchover times using the classical mean-delay results collected by Takagi [113], implemented in `jline/api/polling/`.

Caches. `Solver_mva_cache_analyzer` computes cache hit and miss probabilities via fixed-point iterations over the cache MVA equations, using the `CACHE` API algorithms (Che-style characteristic-time approximations [31] and list-based replacement models [39]); `Solver_mva_cacheqn_analyzer` embeds these probabilities into the surrounding queueing network by class switching, iterating between the cache solution and the network solution until the miss rates converge. The `Solver_mva_retrieval_analyzer` extends this scheme to delayed-hit retrieval caches, in which misses trigger a back-end fetch and concurrent requests for the same item coalesce.

Open network decomposition. The `qna` method (`Solver_qna`) implements two-moment parametric decomposition for open networks of G/G/1 and G/G/k queues in the style of the Queueing Network Analyzer

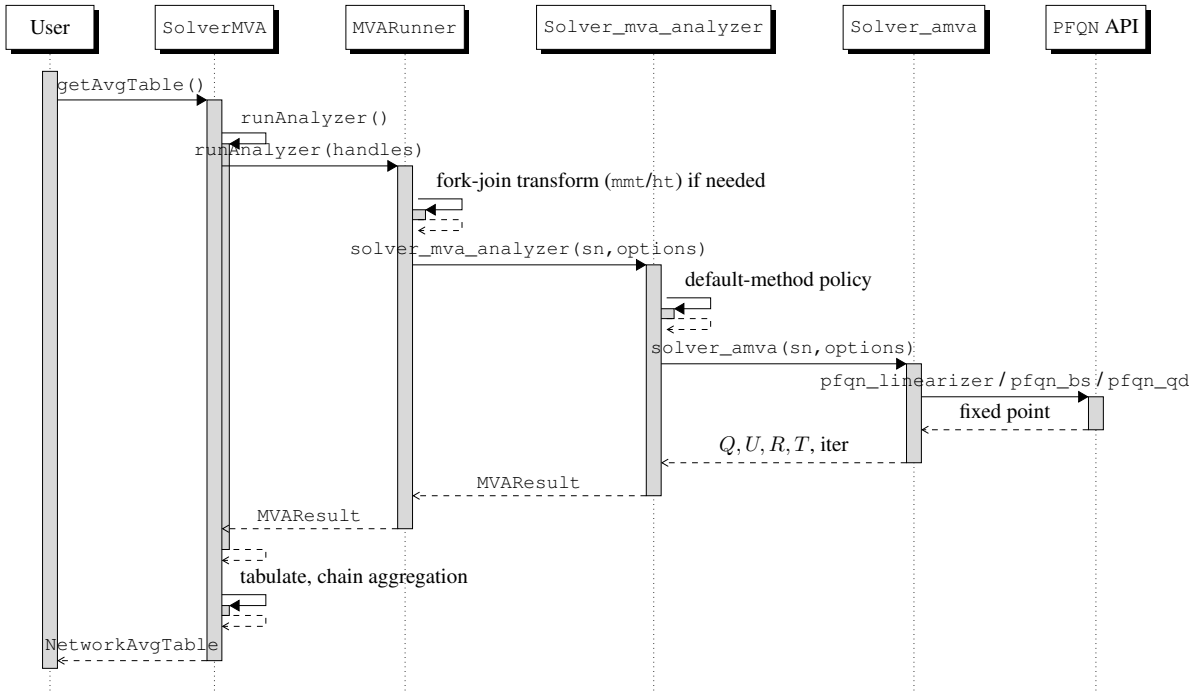

Figure 2.1: Sequence diagram of `SolverMVA` for a default steady-state request.

of Whitt [120], with the Kraemer and Langenbach-Belz waiting-time approximation [72] at each station, and flow superposition, splitting, and departure SCV propagation following [85, 120].

2.8 Fork-join transformations

`MVARunner` recognizes fork-join subnetworks and applies model transformations before invoking the MVA algorithms: the `ht` (`heidelberger-trivedi`) method implements the classical asynchronous-task transformation of Heidelberger and Trivedi [57], while `mmt` (the default) and `fjt` implement mixed-model transformations that replace fork-join sections with auxiliary open classes calibrated iteratively [41]. The transformation loop re-invokes the analyzer on each transformed model until the synchronization delays stabilize.

2.9 Architectural flow

Figure 2.1 shows the sequence of calls for a steady-state average request with the default method on a closed non-product-form network, the most common execution path.

2.10 Implementation notes

The exact and approximate algorithms operate on demands D_{ir} obtained from `sn` as V_{ir}/μ_{ir} after chain-level visit computation (`sn.visits`); class-to-chain aggregation and disaggregation are performed by the `SN` API so that all PFQN algorithms see one class per chain. Numerical safeguards include demand perturbation to break exact ties among stations (which can stall Linearizer convergence), utilization capping at 1 for unstable open models, and the DTMC visit solver ordering (`dtmc_solve` primary, reducible fallback) that guarantees consistent visit ratios in fork-join models. Iteration counts and the achieved residual are reported in `MVAResult` for reproducibility.

2.11 Feature and method support

Table 2.2 maps modelling features to the `SolverMVA` methods. A cell is marked • when the method admits and handles the feature and ○ when it does so approximately or only conditionally; a blank cell means the feature is not admitted by that method or is routed to a different analyzer. Columns `cache` and `cacheqn` denote the standalone cache analyzer (`Solver_mva_cache_analyzer`) and the cache-queueing-network analyzer (`Solver_mva_cacheqn_analyzer`, which also drives delayed-hit retrieval), and `poll` the polling analyzer; all three are selected automatically by topology. Per the method-aware gate, `qna` and `rqna` are open-only (they drop closed and self-looping classes), and `rqna` additionally admits Markovian arrival processes; `sqd` and the bound methods are single-chain closed. Exact MVA is exact for phase-type service under insensitive disciplines (PS, LCFS-PR, INF) but approximate for FCFS with non-exponential service; AMVA is approximate in general.

Table 2.2: Feature support of `SolverMVA` methods (• supported, ○ approximate/conditional, blank unsupported).

Feature	exact	amva	qna	rqna	sqd	bounds	cache	cacheqn	poll
Job classes									
Open class	•	•	•	•			•	•	•
Closed class	•	•			•	•	•	•	•
Self-looping class	•	•			•	•	•	•	•
Mixed open/closed	•	•						•	
Class switching	•	•	•	•			•	•	
Priority classes (HOL)		○							
Node types									
Source, Sink	•	•	•	•			•	•	•
Queueing station	•	•	•	•	•	•		•	•
Delay (infinite server)	•	•	•	•		•		•	
Multiserver ($c > 1$)	•	•	•	○		○			
Fork / Join	•	•							
Cache node							•	•	
Service and arrival distributions									
Exponential	•	•	•	•	•	•	•	•	•
Phase-type (Erlang, HyperExp, Coxian, APH)	•	•	•	•	○	•	•	•	○

continued on next page

Feature	exact	amva	qna	rqna	sqd	bounds	cache	cacheqn	poll
General two-moment (Det, Uniform, Pareto, Weibull, Lognormal)	○	○	●	●	○	○	○	○	○
Batch (BMAP)	○	○	○	○					
Markovian (MAP, MMPP2, MMAP, RAP)				●					
Scheduling disciplines									
INF (delay)	●	●	●	●		●		●	
FCFS	○	●	●	●	●	●		●	
PS	●	●				●		●	
DPS		○							
LCFS, LCFS-PR	●	●				●		●	
SIRO	●	●				●		●	
POLLING									●
Routing strategies									
Probabilistic (PROB)	●	●	●	●	●	●	●	●	●
Random (RAND)	●	●	●	●	●	●	●	●	●
Cache and advanced station features									
Cache replacement (RR, FIFO, LRU)							●	●	
Delayed-hit retrieval							○	●	
Load dependence	●	●							
Finite-capacity blocking (BAS)					●				
Analysis and output									
Steady-state means (`getAvg`)	●	●	●	●	●	●	●	●	●
Bounds (upper/lower)						●			

2.12 Method algorithms

Table 2.3 gives high-level pseudocode for each `SolverMVA` method.

Table 2.3: High-level pseudocode of `SolverMVA` methods.

Method	Pseudocode
`exact`	1. for each population $\boldsymbol{n}$ from $\mathbf{0}$ to $\boldsymbol{N}$ (lexicographic): 2. $R_{ir} = D_{ir}\,[1 + Q_i(\boldsymbol{n} - \mathbf{1}_r)]$ at queues, $R_{ir} = D_{ir}$ at delays (arrival theorem) 3. $T_r = n_r/(Z_r + \sum_i R_{ir})$; $Q_{ir} = T_r R_{ir}$ 4. return Q, U, R, T at $\boldsymbol{n} = \boldsymbol{N}$ (multiserver: add marginal-probability recursion; mixed: inflate closed demands by $(1 - U_i^{\text{open}})^{-1}$)
`amva`	1. initialize $Q_{ir} = N_r/M$ 2. repeat: estimate arrival-instant lengths A_{ir} (Bard-Schweitzer $A_{ir} = Q_i - Q_{ir}/N_r$, or Linearizer deviations δ_{irs}) 3. $R_{ir} = D_{ir}(1 + A_{ir})$; $T_r = N_r/(Z_r + \sum_i R_{ir})$; $Q_{ir} = T_r R_{ir}$ 4. until $\max_{ir} \lvert\Delta Q_{ir}\rvert <$ `iter_tol`
`qna`	1. solve traffic equations for per-station arrival rates λ_i 2. propagate SCVs through merge, split, and departure operators to get $c^2_{a,i}$ 3. per station evaluate G/G/k waiting time (Kraemer–Langenbach-Belz) 4. assemble Q, U, R, T (open only)
`rqna`	as `qna`, but propagate MAP/MMPP2 flow descriptors (rate, SCV, lag-1 correlation) instead of two moments; per-station queue solved from the fitted MAP

continued on next page

Method	Pseudocode
`sqd`	1. mark blocking-after-service stations; 2. iterate MVA with blocked-job holding at the downstream station until the blocking probabilities and throughputs converge (single-chain closed)
`bounds`	compute $D_{\max} = \max_i D_i$, $D_{\text{sum}} = \sum_i D_i$, Z; return $X_{\text{up}} = \min(1/D_{\max}, N/(D_{\text{sum}} + Z))$ and the companion lower bound (asymptotic, balanced-job, geometric, or Harel), no iteration
`cache`	1. initialize per-item miss probabilities 2. solve the characteristic time t_C from $\sum_k h_k(t_C) =$ capacity (Che approximation) 3. update item hit probabilities h_k; iterate to fixed point; return hit/miss rates
`cacheqn`	1. solve the network (`exact/amva`) with current hit/miss class-switch probabilities 2. recompute cache hit/miss from the resulting per-item request rates (`cache`) 3. iterate steps 1–2 until miss rates converge (delayed-hit retrieval adds back-end fetch classes)
`poll`	evaluate exhaustive/gated/limited mean waiting time in closed form from utilizations and switchover moments (Takagi)
fork-join (`mmt`/`ht`)	1. replace each fork-join subnetwork by auxiliary open classes (`mmt`) or asynchronous tasks (`ht`) 2. solve the transformed model with the selected method 3. update synchronization delays from sibling response times; repeat until stable

2.13 Bibliographic notes

Exact MVA is due to Reiser and Lavenberg [99], with the load-dependent form in [98] and the mixed-network form in [16]. The AMVA lineage starts with Bard [5] and Schweitzer [107], continues with Linearizer [30] and its cost reductions, the AQL method [122], multiserver extensions [34, 112], and unified templates [38]; queue-dependent AMVA is introduced in [26] and load-dependent factorization in [24]. Balanced job bounds and asymptotic bounds are classical [77]; geometric bounds are from [25] and the Harel bounds from [55]. Open-network decomposition follows Whitt [120] and Kraemer and Langenbach-Belz [72]. Fork-join approximations trace to Heidelberger and Trivedi [57] and, for the mixed-model transformations, to [41]. Polling results are surveyed by Takagi [113], and M/G/1 scheduling analysis by [93, 121].

Chapter 3

SolverNC

3.1 Overview

`SolverNC` encodes the normalizing-constant paradigm for product-form queueing networks. For a closed BCMP network [6] with population $\boldsymbol{N}$, the equilibrium distribution factorizes as

$$\pi(\boldsymbol{n}) = \frac{1}{G(\boldsymbol{N})} \prod_{i=1}^{M} F_i(\boldsymbol{n}_i), \qquad G(\boldsymbol{N}) = \sum_{\boldsymbol{n}:\sum_i \boldsymbol{n}_i = \boldsymbol{N}} \prod_{i=1}^{M} F_i(\boldsymbol{n}_i), \tag{3.1}$$

where F_i are station factors determined by demands and scheduling. All mean and marginal metrics are ratios of normalizing constants evaluated at shifted populations; for instance $T_r(\boldsymbol{N}) = G(\boldsymbol{N} - \mathbf{1}_r)/G(\boldsymbol{N})$. The solver therefore reduces performance evaluation to the computation, exact or asymptotic, of $G(\boldsymbol{N})$, in contrast with `SolverMVA`, which eliminates G altogether. Working with G pays off for models with many classes but few stations, for marginal and joint probabilities, for load-dependent stations, and for models where asymptotic expansions are accurate.

The solver resides in `jline/solvers/nc/`, with the numerical algorithms in `jline/api/pfqn/nc/` (functions prefixed `pfqn_`). All computations are carried out in logarithmic space (`logNormConstAggr`) to avoid overflow, since $G(\boldsymbol{N})$ grows combinatorially in $\boldsymbol{N}$.

3.2 Software architecture

`SolverNC.runAnalyzer()` dispatches to six analyzers: the default `Solver_nc_analyzer`, the load-dependent `Solver_ncld_analyzer`, cache analyzers (`Solver_nc_cache_analyzer`, `Solver_nc_cache_qn_analyzer`, `Solver_nc_retrieval_analyzer`), and the loss-network analyzer `Solver_nc_lossn_analyzer`. Handlers implement metric extraction from normalizing constants: `Solver_nc` (means), `Solver_nc_conv` (convolution), `Solver_nc_marg`/`Solver_nc_margaggr` (marginal probabilities), `Solver_nc_joint`/`Solver_nc_jointaggr` (joint probabilities), `Solver_ncld` (load-dependent means), and `Solver_nc_lcfsqn` (LCFS networks). The method string selects the algorithm used to evaluate G inside `pfqn_nc`, which acts as a facade that also sanitizes demands (`pfqn_nc_sanitize`: removal of replicated stations, zero-demand classes, and scaling for conditioning).

3.3 Exact methods

ca The convolution algorithm of Buzen [18], generalized to multiclass networks [77, 98], computes G by dynamic programming over stations and populations in $O(MR\prod_r(N_r+1))$ time (`pfqn_ca`). It is the default exact fallback and also serves load-dependent stations through its queue-dependent form.

comom The class-oriented method of moments [19, 20] evaluates G by propagating a basis of normalizing constants at neighboring populations through exact linear systems, with complexity polynomial in the total population for a fixed number of classes; it is the method of choice for models with many jobs and few classes (`pfqn_comom`, with repairmen specializations `pfqn_comomrm` and the multiserver variant `pfqn_comomrm_ms`). The `comomld` variant extends the basis propagation to load-dependent stations.

mom The general method of moments [20] (`pfqn_mom`), based on moment recursions on G; `pfqn_mmsample2` provides sampled cross-checks.

kt The Koenigsberg-Tier style recursion implemented as the Knessl-Tier asymptotic-exact hybrid [67] (`pfqn_kt`).

rd, nrp, nrl Exact recursions on generating functions and partial fractions in the tradition of Harrison and Lee [56] and residue methods [9, 32].

3.4 Asymptotic and integral methods

For large populations, (3.1) admits integral representations

$$G(\boldsymbol{N}) \;=\; \frac{1}{\prod_r N_r!}\int_{\mathbb{R}_+^M}\prod_r\Big(\textstyle\sum_i D_{ir}u_i + Z_r\Big)^{N_r} e^{-\sum_i u_i}\,d\boldsymbol{u}, \tag{3.2}$$

in the style of McKenna and Mitra [86], which the following methods evaluate:

le Logistic expansion [24]: a change of variables maps (3.2) onto the simplex, where a Laplace approximation around the fixed point of the logistic transform gives an asymptotic expansion with computable correction terms (`pfqn_le`, with fixed-point algorithms `pfqn_le_fpi`, `pfqn_le_fpiZ` and Hessian terms `pfqn_le_hessian`, `pfqn_le_hessianZ`). `pfqn_lap` provides the plain Laplace approximation.

ls Logistic sampling [24]: Monte Carlo integration under the logistic change of variables, asymptotically unbiased with variance vanishing in the large-population limit (`pfqn_ls`).

mci, imci Plain and importance-sampling Monte Carlo integration of (3.2) (`pfqn_mci`), in the spirit of the sampling estimators for closed networks [21].

panacea The asymptotic expansion of PANACEA [86] for networks with delays, valid in the normal-usage regime.

mmint2 Two-dimensional repairmen-type integral evaluated by Gauss-Legendre or Gauss-Laguerre quadrature (`pfqn_mmint2_gausslegendre`, `pfqn_mmint2_gausslaguerre`); `gleint` is the general Gauss-Legendre integration path.

gm, grm Grundmann-Moeller simplicial cubature applied to the simplex form of (3.2) (`pfqn_grnmol`); `cub` selects general cubature (`pfqn_cub`).

clw The Choudhury-Leung-Whitt inversion of the generating function of G [32], including its load-dependent form (`pfqn_clw`, `pfqn_clw_lld`).

The default method chooses adaptively: exact recursions for small models, `comom` when the class structure permits, and logistic expansion or sampling for large populations, with the decision logic centralized in `pfqn_nc`.

3.5 Probability and specialized analyzers

Because `SolverNC` manipulates G directly, it exposes exact state probabilities that mean-value methods cannot produce: `getProbAggr` evaluates marginal probabilities $\Pr[\boldsymbol{n}_i]$ as ratios of normalizing constants with station i removed (`Solver_nc_marg`), and `getProbSysAggr` evaluates joint states (`Solver_nc_joint`), including load-dependent variants. The loss-network analyzer solves multi-rate loss networks by the Erlang fixed-point approximation [66] (`Solver_nc_lossn_analyzer`). Cache analyzers compute exact or asymptotic hit probabilities for capacity-constrained caches by normalizing-constant methods over the cache state space, embedding them in queueing networks by the same class-switching iteration used in `SolverMVA`; the theory underlying the cache normalizing constants follows time-to-live and characteristic-time equivalences [31, 39].

3.6 Architectural flow

Figure 3.1 shows the standard path. Note the two-level dispatch: the analyzer selects the handler by request type (means, marginals, joint), while the `pfqn_nc` facade selects the numerical algorithm by method and model size.

3.7 Implementation notes

All algorithms return $\log G$ and the handlers form metric ratios by exponentiating differences, which keeps the pipeline stable up to populations of several thousands. Demand sanitization eliminates numerically troublesome structure before algorithm selection: zero-demand classes are absorbed into think times, identical stations are merged with multiplicities, and demands are rescaled with the scaling factor restored in $\log G$ afterwards. The load-dependent path stores station factors as cumulative log-products of rate scalings, consistent with the factorization form of [24]. Results include `logNormConstAggr` so that downstream consumers, notably `SolverLN` and the sensitivity API (`pfqn/sens`), can reuse the constant without recomputation.

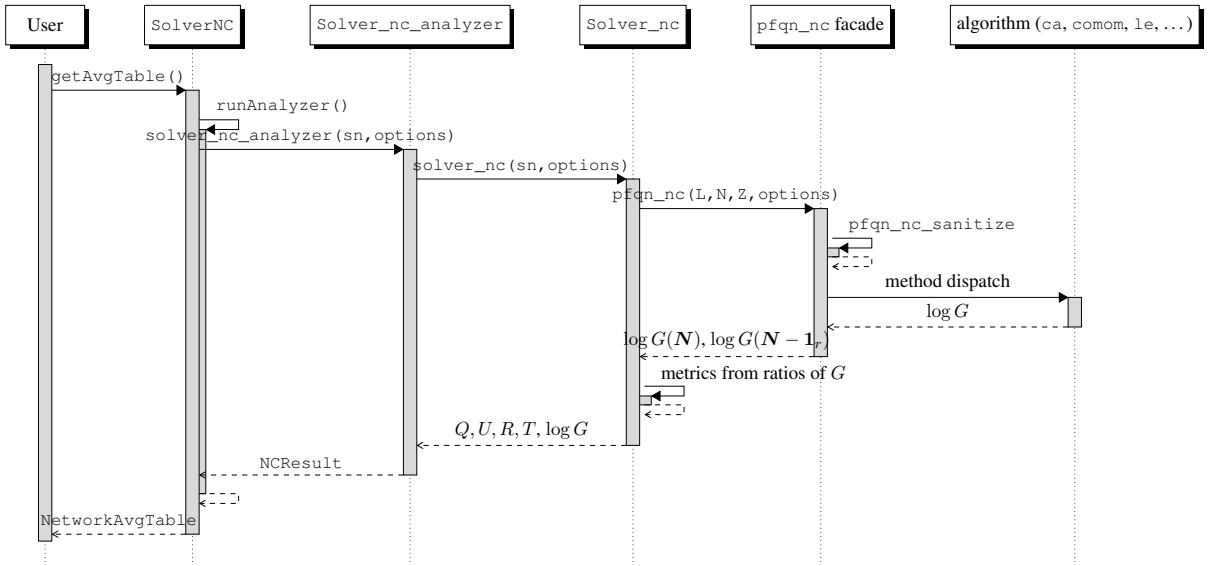

Figure 3.1: Sequence diagram of `SolverNC` for a steady-state average request.

3.8 Feature and method support

Table 3.1 maps modelling features to the `SolverNC` methods. Exact normalizing-constant evaluation (`ca`, `comom`, `mom`) requires product-form structure; the asymptotic and integral methods (`le`/`ls`, `mci`, `kt`) target large closed populations; `mem` is the maximum-entropy path for non-product-form open and closed networks, gated per model class composition (`supportsModelMethod`). Column `spec.` denotes the specialized analyzers (cache, retrieval, loss network), selected by topology. Phase-type service is exact under insensitive disciplines (PS, LCFS-PR, INF) and FCFS is exact only with class-independent exponential service (BCMP), hence ○.

Table 3.1: Feature support of `SolverNC` methods (● supported, ○ approximate/conditional, blank unsupported).

Feature	ca	comom	mom	le/ls	mci	kt	mem	spec.
Job classes								
Open class	●	●	●		●		●	●
Closed class	●	●	●	●	●	●	●	●
Self-looping class	●	●	●	●	●	●		●
Mixed open/closed	●	●	○		●		●	●
Class switching	●	●	●	●	●	●		●
Node types								
Source, Sink	●	●	●		●		●	●
Queueing station	●	●	●	●	●	●	●	●
Delay (infinite server)	●	●	●	●	●	●	●	●

continued on next page

Feature	ca	comom	mom	le/ls	mci	kt	mem	spec.
Cache node								●
Service distributions								
Exponential	●	●	●	●	●	●	●	●
Phase-type (Erlang, HyperExp, Coxian, APH)	●	●	●	●	●	●	○	●
General two-moment (Det)	○	○	○	○	○		○	○
Scheduling disciplines								
INF (delay)	●	●	●	●	●	●	●	●
PS	●	●	●	●	●	●	●	●
LCFS-PR	●	●	●	●	●	●	●	●
SIRO	●	●	●	●	●	●		●
FCFS	○	○	○	○	○	○	●	○
Advanced and specialized features								
Load dependence	●	●		●				
Cache replacement (RR, FIFO)								●
Delayed-hit retrieval								●
Loss network (Erlang fixed point)								●
Analysis and output								
Steady-state means (`getAvg`)	●	●	●	●	●	●	●	●
Marginal probabilities (`getProbAggr`)	●	●	●	○	○			●
Joint probabilities (`getProbSysAggr`)	●	●	●	○	○			
Normalizing constant (`logNormConst`)	●	●	●	●	●	●	●	●

3.9 Method algorithms

Table 3.2 gives high-level pseudocode for each `SolverNC` method; all operate in logarithmic space and return $\log G(\boldsymbol{N})$ and the shifted $\log G(\boldsymbol{N}-\mathbf{1}_r)$.

Table 3.2: High-level pseudocode of `SolverNC` methods.

Method	Pseudocode
`ca`	1. $g \leftarrow \delta_{\boldsymbol{n},\mathbf{0}}$ 2. for each station i: convolve g with the station factor sequence $F_i(\boldsymbol{n}_i)$ over the population lattice 3. $G(\boldsymbol{N}) = g(\boldsymbol{N})$; form metrics as ratios $G(\boldsymbol{N}-\mathbf{1}_r)/G(\boldsymbol{N})$ (load-dependent: queue-dependent factors)
`comom`	1. assemble a basis of normalizing constants at neighboring populations 2. propagate the basis from small to full $\boldsymbol{N}$ through exact linear systems (class-oriented moment equations) 3. read $\log G$ and shifts from the final basis (`comomld`: load-dependent factors)
`mom`	solve moment recursions on G by linear systems over populations; `mmsample2` provides sampled cross-checks
`le`	1. map the integral to the simplex via the logistic change of variables 2. solve the fixed point $\boldsymbol{u}^*$ of the logistic transform 3. Laplace expansion around $\boldsymbol{u}^*$ with Hessian correction terms $\Rightarrow \log G$
`ls`	Monte Carlo integration under the logistic change of variables; average the integrand over samples; variance vanishes in the large-population limit
`mci`	draw $\boldsymbol{u} \sim \exp$; average $\prod_r (\sum_i D_{ir} u_i + Z_r)^{N_r}$; `imci` adds importance sampling
`kt`	Knessl–Tier asymptotic-exact hybrid recursion on G for large populations

continued on next page

Method	Pseudocode
`mem`	1. form the maximum-entropy product-form marginal per station with Lagrange multipliers 2. match multipliers to mean queue length / utilization constraints 3. iterate global flow balance until the marginals are consistent (non-product-form, open and closed)
`spec.`	cache: normalizing-constant / characteristic-time fixed point over cache states; loss network: Erlang fixed-point (Kaufman–Roberts) recursion over occupancy

3.10 Bibliographic notes

Product-form theory originates with Jackson [63], Gordon and Newell [52], and BCMP [6]; the convolution algorithm is due to Buzen [18] with multiclass and load-dependent treatments in [77, 98]. RECAL and related recursions by chains are surveyed in [33, 35]. Integral representations and asymptotics were developed by McKenna and Mitra [86], Knessl and Tier [67], and via residues and transform inversion in [9, 32, 56]. The method of moments and its class-oriented specialization are from [19, 20]; logistic expansion and sampling from [24]; asymptotic performance inference applications in [21]. Loss-network fixed points follow Kelly [66]. A unified treatment of normalizing-constant methods appears in [12].

Chapter 4

SolverCTMC

4.1 Overview

`SolverCTMC` evaluates a model by explicit construction of its underlying continuous-time Markov chain: it enumerates the reachable state space, assembles the infinitesimal generator $\boldsymbol{Q}$, and solves the global balance equations $\boldsymbol{\pi Q} = \mathbf{0}$, $\boldsymbol{\pi 1} = 1$ for steady-state metrics, or the Kolmogorov forward equations for transient metrics [73, 111]. It is the semantic ground truth of LINE: every extended feature of the modeling language (phase-type services, class switching, impatience, retrials, finite capacity regions, Petri net primitives, heterogeneous servers, signals) is ultimately defined by the CTMC that this solver builds, and other solvers are validated against it. Its practical limitation is state-space explosion, so it is intended for models whose reachable space fits in memory, with a configurable cutoff (`options.cutoff`) truncating open models.

The solver resides in `jline/solvers/ctmc/`, with state-space and Markov-chain algorithms in `jline/api/mc/` and the state marshalling machinery in `jline/lang/state/`.

4.2 State-space generation

The state of a stateful node is a vector encoding, per class, the queue contents (with buffer positions for order-sensitive disciplines), the service phases of in-service jobs, and local variables (routing pointers, cache lists, Petri net markings), following the state descriptors documented in the `NetworkStruct` reference. Two generation engines exist:

synchronization-based (default) Events are described by the `sn` synchronization structures (`sn.sync`): each synchronization couples an active action (e.g., a departure at node i in class r) with a passive action (the matching arrival). The generator `Ctmc_ssg` performs a reachability exploration from the initial state, applying `afterEvent` state-transition functions node by node; rates are composed from the active action rate and the passive routing probability. Fork-join models use the tag-augmented generator `Ctmc_ssg_fj`, which introduces transient tag classes tracking sibling tasks so that join synchronization is Markovian.

flat A legacy engine that enumerates the Cartesian product of node state spaces and filters unreachable

states; it is retained as a fallback for features not yet covered by the synchronized generator, e.g., certain MAP compositions and round-robin routers.

Immediate transitions (zero service times, Petri net immediate firings, pass-through routers) are eliminated by stochastic complementation [87] (`Ctmc_stochcomp`, `Ctmc_pseudostochcomp`): the generator is partitioned into tangible and vanishing states and the vanishing block is absorbed, as in generalized stochastic Petri net reductions [1]. A `MemoryGuard` component estimates the state-space cardinality before allocation and aborts with a diagnostic when the projected memory exceeds the configured budget.

4.3 Steady-state and transient solution

The generator is assembled by `Ctmc_makeinfgen` in sparse form. The steady-state algorithm `Ctmc_solve` orders methods by robustness: direct sparse LU on the augmented balance equations, with diagonal scaling; when the chain is reducible or nearly so, `Ctmc_solve_reducible` performs a block decomposition into communicating classes and solves each recurrent block (`Ctmc_solve_reducible_blkdecomp`). Iterative aggregation-disaggregation is available through the Takahashi method [69, 114] (`Ctmc_takahashi`) and Courtois decomposition for nearly completely decomposable chains [36, 37] (`Ctmc_courtois`); `Ctmc_kms` implements the Koury-McAllister-Stewart scheme. Kolmogorov reversibility testing (`Ctmc_testpf_kolmogorov`) and time reversal (`Ctmc_timereverse`) support product-form verification experiments [65].

Transient analysis (`getTranAvg`, `getTranProb`) integrates $\frac{d}{dt}\boldsymbol{\pi}(t) = \boldsymbol{\pi}(t)\boldsymbol{Q}$ by uniformization [64, 111] (`Ctmc_randomization`, `Ctmc_transient`): with $\Lambda \geq \max_x |q_{xx}|$ and $\boldsymbol{P} = \boldsymbol{I} + \boldsymbol{Q}/\Lambda$,

$$\boldsymbol{\pi}(t) = \sum_{n \geq 0} e^{-\Lambda t} \frac{(\Lambda t)^n}{n!} \boldsymbol{\pi}(0) \boldsymbol{P}^n, \tag{4.1}$$

truncated adaptively to the requested tolerance. Time-averaged distributions used by reward and availability analysis are computed by `Ctmc_timeaverage`.

4.4 Analyzers and methods

The default analyzer `Solver_ctmc_analyzer` runs the pipeline above and derives station-class metrics from $\boldsymbol{\pi}$ through `Ctmc_avg_from_pi`, which projects state probabilities onto per-station, per-class counting functions; utilizations account for multiserver and heterogeneous-server stations by counting busy servers per state. Marginal and joint probability requests route to the handlers `Solver_ctmc_marg`, `Solver_ctmc_margaggr`, `Solver_ctmc_joint`, and `Solver_ctmc_jointaggr`, which differ in whether phases are kept or aggregated. Reward analysis (`getAvgReward`, `Solver_ctmc_reward`) evaluates $\mathbb{E}[\rho] = \sum_x \pi_x \, \rho(x)$ for user-supplied state functions ρ registered in `sn.reward`, supporting nonlinear functionals such as $\mathbb{E}[n^2]$, in the tradition of Markov reward models [87, 104].

The `qrf` method invokes `Solver_ctmc_qrf_analyzer`, an optimization-based evaluation of complex stochastic networks (Quadratic Reduction Framework) that replaces full enumeration with a constrained optimization over aggregated state variables [23]. The `gpu` method offloads the linear-algebra algorithms

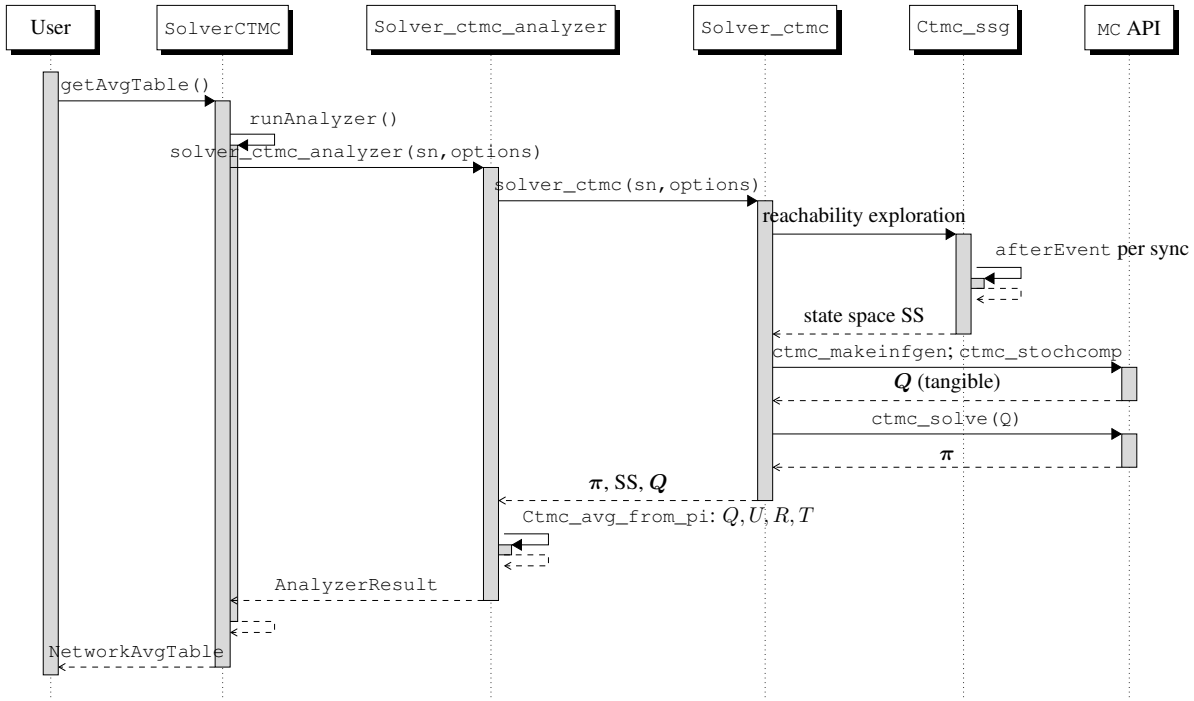

Figure 4.1: Sequence diagram of `SolverCTMC` for a steady-state average request with the synchronization-based generator.

to GPU backends when available. Stochastic Petri net models (Place/Transition nodes) are evaluated on the same pipeline, with markings as states and timed transitions as synchronizations, following GSPN semantics [1, 88, 90]; non-Markovian firing distributions (deterministic, Gamma, Weibull, lognormal, Pareto, uniform) are expanded into phase-type approximations before generation [11].

4.5 Architectural flow

Figure 4.1 shows the standard path. Transient requests replace `ctmc_solve` with uniformization over the same generator; probability requests replace the projection step with the marginal or joint handlers.

4.6 Implementation notes

State vectors are canonicalized and hashed so that reachability exploration is a breadth-first search with $O(1)$ duplicate detection; `sn` index-conversion methods map node states to station metrics. The solver can export the symbolic generator (`getSymbolicGenerator`) with transition labels, supporting derivation of parametric chains, and can save the state space for inspection (`getStateSpace`, `ctmc_simulate` for sample-path cross-checks). Open classes require a finite cutoff; the truncation error is not bounded by the solver and should be assessed by cutoff refinement. The analyzer caps reported utilizations at 1 for unstable

open models and emits a warning, consistently with the tool-wide convention. For models with cache nodes, hit and miss probabilities are exact functionals of π, making the solver the reference for cache-analysis validation.

4.7 Feature and method support

Because `SolverCTMC` builds the exact chain, feature admission does not vary by solution algorithm: the steady-state methods (direct sparse solve, `Ctmc_solve_reducible`, Takahashi, Courtois, KMS, GPU offload) and the transient uniformization path all operate on the same generator and therefore share the feature envelope of Table 4.1. The methods differ in numerics and in output: transient averages and probabilities (`getTranAvg`, `getTranProb`) use uniformization, marginal and joint probabilities use the `marg`/`joint` handlers, and reward functionals use `Solver_ctmc_reward`; all are available for every feature in the envelope. The `qrf` optimization method covers a structural subset. Non-Markovian firing and service distributions are admitted through phase-type expansion, hence ○.

Table 4.1: Feature envelope of `SolverCTMC` (● supported, ○ supported via phase-type expansion).

Feature	CTMC	**Feature**	CTMC
Open / closed / self-looping class	●	Class switching	●
Priority classes	●	Signal / catastrophe	●
Source, Sink, Queue, Delay	●	Router	●
Fork / Join	●	Cache node	●
Place, Transition (timed/immediate)	●	Enabling / inhibitor arc	●
Exponential	●	Phase-type (Erlang, HyperExp, Coxian, APH, PH)	●
MAP, MMPP2	●	General (Gamma, Weibull, Lognormal, Pareto, Uniform, Det)	○
INF, FCFS, PS	●	DPS, GPS	●
LCFS, LCFS-PR	●	SEPT, LEPT	●
SIRO, HOL	●	LPS	●
PAS (pass-and-swap)	●	Priority variants (`*PRIO`)	●
Routing PROB, RAND	●	RROBIN, WRROBIN	●
JSQ, KCHOICES	●	Cache repl. (RR/FIFO/SFIFO/LRU)	●
Load dependence	●	Retrial (orbit)	●
Balking	●	Reneging (impatience)	●
Steady means (`getAvg`)	●	Marginal/joint prob.	●
Transient (`getTranAvg/Prob`)	●	Rewards (`getAvgReward`)	●
Symbolic generator	●	State-space export	●

4.8 Method algorithms

All methods share the generation stage of Table 4.2 and differ in how the generator is solved.

Table 4.2: High-level pseudocode of `SolverCTMC` stages and methods.

Stage / method	Pseudocode
generation	1. BFS from the initial state; for each synchronization apply `afterEvent` node-by-node to reach successors, hashing states for $O(1)$ deduplication 2. assemble sparse $\boldsymbol{Q}$ (`makeinfgen`); eliminate immediate/vanishing states by stochastic complementation
`default` (direct)	solve $\boldsymbol{\pi Q} = \mathbf{0}$, $\boldsymbol{\pi 1} = 1$ by sparse LU on the augmented, diagonally scaled system
reducible	decompose into communicating classes; solve each recurrent block and weight by reachability
Takahashi / Courtois / KMS	iterative aggregation–disaggregation: partition states, solve the aggregated chain, disaggregate, iterate to convergence (NCD chains)
transient	uniformize with $\Lambda \geq \max_x \lvert q_{xx}\rvert$, $\boldsymbol{P} = \boldsymbol{I} + \boldsymbol{Q}/\Lambda$; $\boldsymbol{\pi}(t) = \sum_{n\geq 0} e^{-\Lambda t}\frac{(\Lambda t)^n}{n!}\boldsymbol{\pi}(0)\boldsymbol{P}^n$, truncated to tolerance
reward	$\mathbb{E}[\rho] = \sum_x \pi_x\, \rho(x)$ for each registered state functional ρ (steady or time-averaged)
`qrf`	replace enumeration by constrained optimization over aggregated state variables (quadratic reduction)
metrics	project $\boldsymbol{\pi}$ onto per-station, per-class counting functions (`avg_from_pi`); marginal/joint handlers keep or aggregate phases

4.9 Bibliographic notes

Numerical solution of Markov chains is treated comprehensively by Stewart [111]; uniformization goes back to Jensen [64]. Aggregation-disaggregation methods are due to Takahashi [114] and Koury, McAllister and Stewart [69], and decomposition of nearly completely decomposable chains to Courtois [36, 37]. Stochastic complementation theory is from Meyer [87]. GSPN semantics and vanishing-state elimination follow [1, 40, 88, 90]. Markov reward models are surveyed in [104]. The QRF optimization framework is introduced in [23]. Product-form testing connects to the reversibility theory of Kelly [65] and to algorithmic product-form detection [4, 83, 84].

Chapter 5

SolverSSA

5.1 Overview

`SolverSSA` evaluates models by stochastic simulation of the same continuous-time Markov chain that `SolverCTMC` builds explicitly, without ever materializing the state space. It implements exact-sampling algorithms in the lineage of Gillespie's stochastic simulation algorithm (SSA) [51]: from the current state, the set of enabled events and their rates is computed, the sojourn time is sampled from the exponential distribution with the total exit rate, and the next event is drawn proportionally to its rate. Metrics are time averages over the generated trajectory. The solver therefore shares the full feature coverage of the CTMC semantics while scaling to state spaces far beyond enumeration, at the cost of statistical rather than numerical error.

The solver resides in `jline/solvers/ssa/`. Unlike `SolverLDES` (Chapter 8), which is an event-driven discrete-event simulator operating on job entities, `SolverSSA` samples Markov jump processes at the state-vector level, which makes it exact for the model semantics but restricted to Markovian (phase-type expanded) timing.

5.2 Analyzers and simulation engines

`Solver_ssa_analyzer` dispatches on `options.method` to four engines:

`serial` (default) The direct-method engine `Solver_ssa`. Each step calls `Solver_ssa_findenabled` to enumerate enabled synchronizations from `sn` (`sn.sync`), evaluates their rates via the same `afterEvent` transition functions used by the CTMC generator, and samples the next event by inverse transform. Hashed states accumulate sojourn times so that stationary metrics are computed from the empirical occupancy measure; an optional event log preserves the sample path for trace-driven post-processing.

`parallel` (alias `para`) The replication engine `Solver_ssa_analyzer_parallel` runs independent trajectories with distinct substreams and averages their stationary estimators, reducing wall-clock time on multicore hosts; the estimator is the across-replication mean, following standard independent-replication methodology [57, 78].

nrm The next-reaction-method engine `Solver_ssa_nrm`, after Gibson and Bruck [50]: each synchronization maintains a tentative firing time in an indexed priority queue, and only the rates affected by the executed event are resampled, using a dependency graph between synchronizations. This reduces the per-event cost from $O(E)$ to $O(\log E)$ for models with E synchronizations and sparse interactions.

nrm.space A state-space-aware NRM variant (`Solver_ssa_nrm_space`) that additionally tracks the visited-state table for reachability and probability output (`Solver_ssa_reachability`), effectively providing simulation-based state-space exploration for models too large for `Ctmc_ssg`.

Tau-leaping acceleration of Markovian queueing simulation, which inspired parts of this architecture, is studied in [109]; the production engines in LINE sample exactly and do not leap.

5.3 Statistical output analysis

The trajectory length is controlled by `options.samples` and the initial transient is removed by discarding a configurable warm-up fraction. Steady-state estimators are ratio estimators over the retained horizon: utilizations and mean queue lengths integrate state indicator functions against sojourn times, throughputs count event firings per unit time, and response times follow from Little's law applied at station level [82]. Replicated runs (parallel engine) report confidence intervals from the sample variance of replication means. The random number generator is seeded from `options.seed`, and all engines derive per-replication substreams deterministically from it, making runs bit-reproducible across the codebases for a fixed seed.

5.4 Architectural flow

Figure 5.1 shows the serial path; the NRM engines replace the per-step rate enumeration with priority-queue updates driven by the synchronization dependency graph.

5.5 Implementation notes

The engines snapshot and restore any `sn` fields they must specialize (e.g., effective server counts for infinite-server stations), so that the shared structure is never mutated across calls. Immediate-feedback suppression (`sn.immfeed`) short-circuits departures that re-enter the same station in the same class, matching the CTMC reduction of vanishing states. Fractional routing stoichiometry is disallowed: events fire with integral job movements, and the NRM dependency graph is rebuilt whenever class switching changes the enabling structure. For models with cache nodes, the engines update the cache list variables inside `afterEvent`, so hit/miss ratios are simulated exactly rather than approximated. The serial engine is the reference implementation; NRM engines are validated against it in the parity suite.

5.6 Feature and method support

`SolverSSA` samples the same Markov chain as `SolverCTMC`, so its feature envelope (Table 5.1) does not vary across the simulation engines: `serial`, `parallel`, `nrm`, and `nrm.space` accept the same models

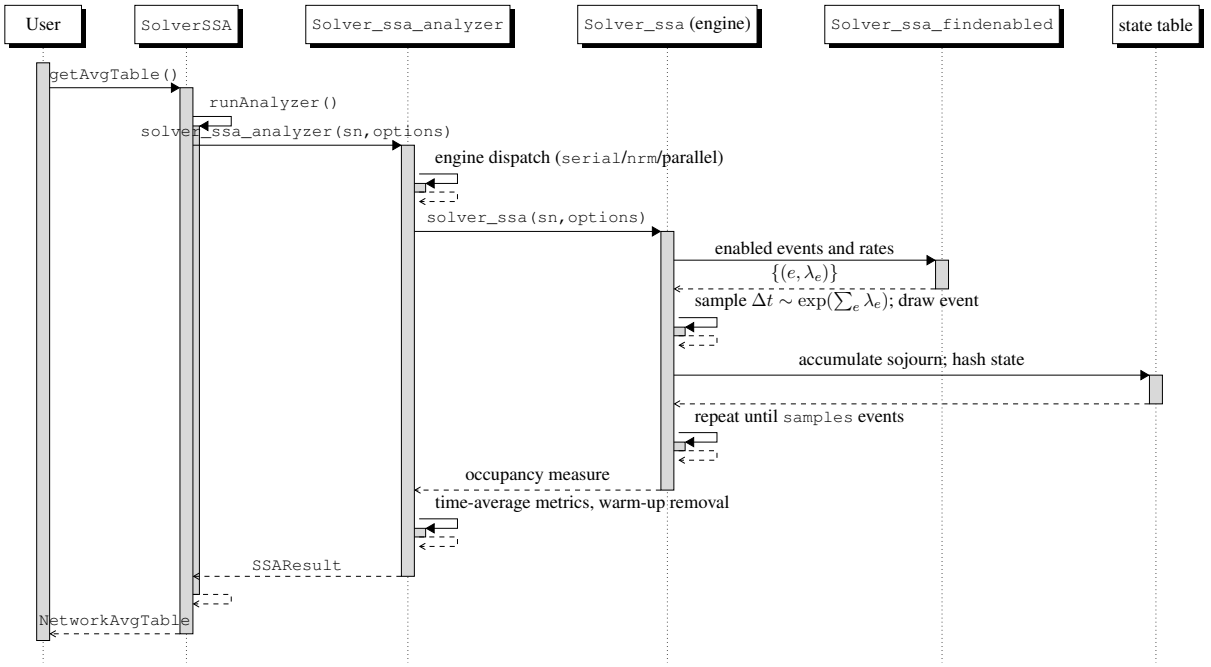

Figure 5.1: Sequence diagram of `SolverSSA` with the serial direct-method engine.

and differ only in cost and in output. The `parallel` engine adds replication confidence intervals, and `nrm.space` additionally exposes reachability and state probabilities. Non-Markovian distributions enter through phase-type expansion (○). Unlike CTMC, SSA also admits reinforcement-learning routing and external (EXT) scheduling.

Table 5.1: Feature envelope of `SolverSSA` (● supported, ○ via phase-type expansion).

Feature	SSA	**Feature**	SSA
Open / closed / self-looping class	●	Class switching	●
Priority classes	●	Source, Sink, Queue, Delay	●
Router	●	Fork / Join	●
Cache node	●	Cache repl. (RR/FIFO/SFIFO/LRU)	●
Exponential	●	Phase-type (Erlang, HyperExp, Coxian, APH, PH)	●
MAP, MMPP2	●	General (Gamma, Weibull, Lognormal, Pareto, Uniform, Det)	○
INF, FCFS, PS	●	DPS, GPS, LPS	●
LCFS, LCFS-PR	●	SEPT, LEPT	●
SIRO, HOL	●	PAS (pass-and-swap)	●
EXT (external self-loop)	●	Priority variants (`*PRIO`)	●
Routing PROB, RAND	●	RROBIN, WRROBIN	●
JSQ, KCHOICES	●	RL (reinforcement learning)	●
Load dependence	●	Retrial (orbit)	●
Balking	●	Reneging (impatience)	●
Steady means (`getAvg`)	●	Confidence intervals (parallel)	●
Probabilities (`nrm.space`)	●	State-space reachability (`nrm.space`)	●

5.7 Method algorithms

Table 5.2 gives high-level pseudocode for each simulation engine.

Table 5.2: High-level pseudocode of `SolverSSA` engines.

Engine	Pseudocode
`serial`	1. from the current state enumerate enabled synchronizations and rates $\{(e, \lambda_e)\}$ (`findenabled`) 2. sample sojourn $\Delta t \sim \exp(\sum_e \lambda_e)$; draw event e with probability $\lambda_e / \sum \lambda_e$ 3. apply `afterEvent`; accumulate sojourn against the hashed state 4. repeat until `samples` events; time-average the occupancy measure after warm-up removal
`parallel`	run R independent trajectories on distinct substreams; average the per-replication stationary estimators; report confidence intervals from their variance
`nrm`	maintain a tentative firing time per synchronization in an indexed priority queue; fire the minimum; resample only the rates affected by the executed event (dependency graph), $O(\log E)$ per step
`nrm.space`	as `nrm`, additionally recording the visited-state table for reachability and state-probability output

5.8 Bibliographic notes

The direct method is due to Gillespie [51]; the next reaction method to Gibson and Bruck [50]. Tau-leaping applied to queueing networks is explored in TauSSA [109]. General discrete-event and replication methodology can be found in [78]. The synchronization formalism that both `SolverSSA` and `SolverCTMC` interpret descends from the event-based semantics of stochastic process algebras and stochastic automata networks [58, 111], adapted to the extended queueing network features of LINE [22].

Chapter 6

SolverFluid

6.1 Overview

`SolverFluid` (`FLD`) analyzes queueing networks by fluid and mean-field approximation: the stochastic population process is replaced by the solution of a system of ordinary differential equations that becomes exact in the many-jobs scaling limit of Kurtz [74, 75]. For a network with phase-type services, the state is the vector $\boldsymbol{x}(t)$ of expected jobs per station, class, and phase, and the ODE system takes the form

$$\dot{\boldsymbol{x}}(t) = \boldsymbol{x}(t)\,\boldsymbol{W}(\boldsymbol{x}(t)), \tag{6.1}$$

where the rate matrix $\boldsymbol{W}$ collects phase-type dynamics, routing, and a scheduling-dependent capacity-sharing term, e.g., $\min(x_i, c_i)/x_i$ for PS multiserver stations. Steady-state metrics are read from the equilibrium $\boldsymbol{x}(\infty)$, and transient metrics from the trajectory; response-time distributions are obtained by augmenting the system with passage-time tracer classes. Fluid limits of this kind for closed multiclass networks with PS and delay stations were derived in [14, 27, 116], and the solver descends from that line of work, with FCFS handled by smoothed approximations [103].

The solver resides in `jline/solvers/fluid/`, with analyzers in `analyzers/` and ODE right-hand sides in `handlers/`. Numerical integration uses stiff and non-stiff solvers from Apache Commons Math, plus an LSODA extension (`LSODAExt`) with automatic stiffness switching.

6.2 Analyzers and methods

`closing` (default) `ClosingAndStateDepMethodsAnalyzer` with the `ClosingAndStateDepMethodsODE` right-hand side. The ODE system (6.1) is built per phase with closing of the service processes: each station's phase-type representation $(\boldsymbol{D}_0, \boldsymbol{D}_1)$ contributes linear intra-phase dynamics and completion flows routed by `sn` routing matrices. Scheduling enters through state-dependent saturation functions; the `softmin` option replaces $\min(x_i, c_i)$ with a smoothed softmin to improve integrability near the corner point, following the smoothing analysis of [103].

`statedep` The same analyzer with fully state-dependent rate scalings, covering load-dependent stations by interpolating `sn.lldscaling` inside $\boldsymbol{W}(\boldsymbol{x})$.

matrix `MatrixMethodAnalyzer` with `MatrixMethodODE`: for networks whose fluid equations are linear except for the capacity term, the right-hand side is assembled once in matrix form, enabling faster evaluation and, in the linear regime, matrix-exponential solutions.

mfq `MFQAnalyzer` evaluates Markov-modulated fluid queues: infinite-buffer fluid models driven by a background CTMC, solved by the spectral and matrix-analytic techniques for stochastic fluid flows [3, 7, 54, 105], with compositional approximations in the style of [43]. This analyzer underpins the `mfq_*` algorithms ported from established fluid-flow toolboxes and connects to fluid stochastic Petri nets [59].

rmf `RMFAnalyzer` implements the refined mean-field approximation of Gast and Van Houdt [49]: the equilibrium of (6.1) is corrected by a $1/N$ expansion term obtained from the Jacobian and the diffusion covariance of the population process, markedly improving accuracy for moderate populations.

Immediate transitions are removed before integration by `ImmediateElimination`, mirroring the vanishing-state reduction of the CTMC solver. The `PStarSearcher` and `PStarOptimisationFunction` components calibrate the blending parameter p^* used to interpolate FCFS stations between PS-like and delay-like fluid regimes, solving a scalar optimization per station [103].

6.3 Transient analysis and passage times

Transient averages (`getTranAvg`) return the integrated trajectory $\boldsymbol{x}(t)$ on the requested time grid, with the initial condition taken from the model state (`sn.state`) or a marginal prior. Response-time distributions (`getCdfRespT`) use `PassageTimeODE`: a tagged infinitesimal tracer population is injected at the reference station and the CDF is read from its absorption dynamics, following the fluid passage-time methodology developed for SLA assessment in [94, 96]. The ODE right-hand side can also be exported symbolically (`FluidODEsExporter`, `exportODEs`) as LaTeX or code, enabling external verification of the generated vector field.

6.4 Architectural flow

Figure 6.1 shows the default path. Stationarity is detected by a step handler (`MethodStepHandler`) monitoring the norm of the derivative; if the integrator stalls, the analyzer switches solver class (non-stiff to stiff) and retries.

6.5 Implementation notes

The three codebases use different ODE integrators (MATLAB `ode15s`/`ode23s`, Apache Commons Math and LSODA in Java, SciPy in Python), so small numerical deviations across implementations are expected and tolerated by the parity suite. The phase-expanded state dimension is $\sum_{i,r}\texttt{phases}(i,r)$, and Jacobians for the stiff solvers are assembled analytically in the matrix method and numerically otherwise. Utilizations are derived from the equilibrium inflow rates rather than from $\boldsymbol{x}(\infty)$ directly, which is more robust when the fluid fixed point lies on the saturation boundary. Known limitations are documented in the feature set:

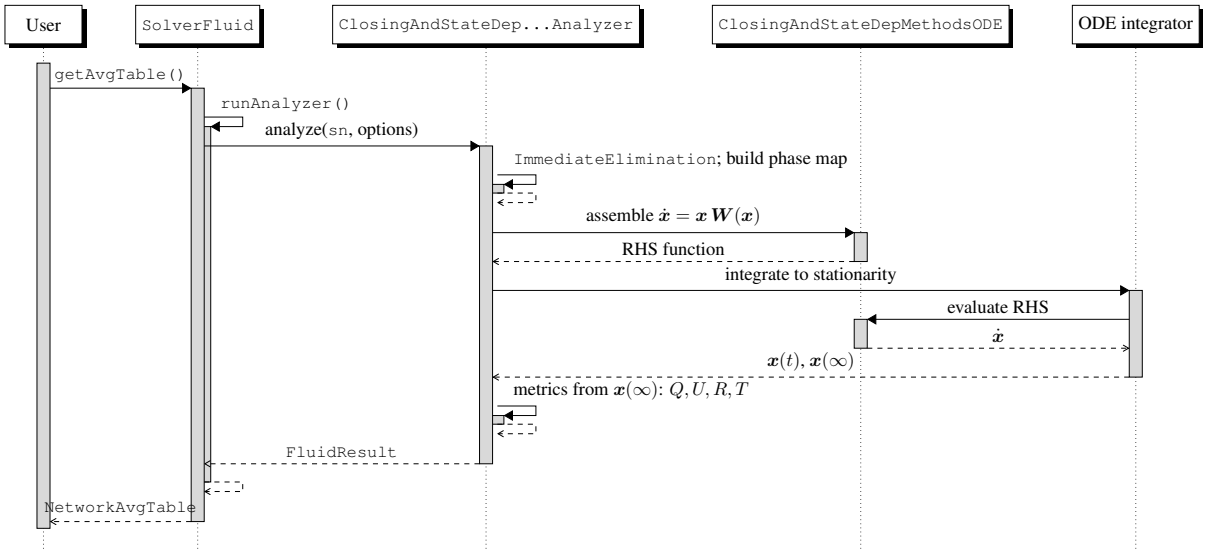

Figure 6.1: Sequence diagram of `SolverFluid` with the default closing method.

multiclass FCFS with highly variable service is approximated through the p^* blending and should be cross-checked against simulation.

6.6 Feature and method support

Table 6.1 maps modelling features to the `SolverFluid` methods. `closing` is the general-purpose analyzer; `statedep` specializes it to state-dependent (load-dependent) scalings; `matrix` assembles the same field once for the linear-plus-capacity regime; `mfq` is the Markov-modulated fluid-queue analyzer, a distinct model class (fluid buffer driven by a background CTMC) rather than a job-population network; and `rmf` adds the refined mean-field $1/N$ correction to the equilibrium of the population ODE. FCFS is captured only through the p^* smoothing and is therefore ○; general distributions enter through phase-type/moment closure (○).

Table 6.1: Feature support of `SolverFluid` methods (● supported, ○ approximate/-conditional, blank unsupported).

Feature	closing	statedep	matrix	mfq	rmf
Job classes					
Open class	●	●	●	○	
Closed class	●	●	●		●
Self-looping class	●	●	●		○
Class switching	●	●	●		○
Node types					

continued on next page

Feature	closing	statedep	matrix	mfq	rmf
Source, Sink	●	●	●	○	
Queueing station	●	●	●	●	●
Delay (infinite server)	●	●	●		●
Cache node	●	●	●		
Service distributions					
Exponential	●	●	●	●	●
Phase-type (Erlang, HyperExp, Coxian, Cox2, APH)	●	●	●	○	●
General (Gamma, Weibull, Lognormal, Pareto, Uniform, Det)	○	○	○		○
Scheduling disciplines					
INF (delay)	●	●	●		●
PS	●	●	●		●
DPS	●	●	●		○
FCFS	○	○	○		○
LCFS, LCFS-PR	●	●	●		○
SIRO	●	●	●		
Advanced features					
Load dependence	○	●	●		
Markov-modulated fluid (background CTMC)				●	
Refined mean-field $1/N$ correction					●
Analysis and output					
Steady-state means (`getAvg`)	●	●	●	●	●
Transient averages (`getTranAvg`)	●	●	●	○	
Response-time CDF (`getCdfRespT`)	●	●	○		
Symbolic ODE export (`exportODEs`)	●	●	●		

6.7 Method algorithms

Table 6.2 gives high-level pseudocode for each `SolverFluid` method.

Table 6.2: High-level pseudocode of `SolverFluid` methods.

Method	Pseudocode
`closing`	1. build the phase-expanded state $\boldsymbol{x}$ and the ODE $\dot{\boldsymbol{x}} = \boldsymbol{x}\,\boldsymbol{W}(\boldsymbol{x})$ from $(\boldsymbol{D}_0, \boldsymbol{D}_1)$, routing, and the capacity term $\min(x_i, c_i)/x_i$ (`softmin` smoothing near the corner) 2. eliminate immediate transitions; integrate to stationarity (step handler on $\|\dot{\boldsymbol{x}}\|$, stiff/non-stiff switch) 3. read Q, U, R, T from $\boldsymbol{x}(\infty)$ (utilizations from inflow rates)
`statedep`	as `closing` with fully state-dependent rate scalings ($\boldsymbol{W}(\boldsymbol{x})$ interpolates `lldscaling`)
`matrix`	assemble $\boldsymbol{W}$ once in matrix form (linear except the capacity term); integrate, using matrix-exponential solution in the linear regime
`mfq`	Markov-modulated fluid queue: solve the fluid buffer driven by the background CTMC by spectral / matrix-analytic techniques for the stationary buffer distribution
`rmf`	1. solve the closing fixed point $\boldsymbol{x}^*$ 2. add the refined mean-field $1/N$ correction from the Jacobian and the diffusion covariance of the population process

6.8 Bibliographic notes

Mean-field limits for Markov population processes are due to Kurtz [74, 75]. Fluid analysis of closed multi-class queueing networks and of PS stations was developed in [14, 27], with reward-based extensions in [116] and model-transformation applications in [89]. Passage-time and percentile analysis via fluid tracers is from [94, 96]. The refined mean-field correction is due to Gast and Van Houdt [49], and the smoothed FCFS fluid model to Ruuskanen et al. [103]. Markov-modulated fluid queues originate with Anick, Mitra and Sondhi [3], with matrix-analytic treatments in [7, 54, 105], compositional approximations in [43], and fluid stochastic Petri nets in [59].

Chapter 7

SolverMAM

7.1 Overview

`SolverMAM` encodes matrix-analytic methods for queueing models with Markovian arrival processes (MAPs) and phase-type or MAP services [76,91,92]. Its core object is the quasi-birth-death (QBD) process: for a MAP/MAP/1 queue, the CTMC on levels (queue lengths) and phases has block-tridiagonal generator with blocks $(\boldsymbol{A}_0, \boldsymbol{A}_1, \boldsymbol{A}_2)$, and the stationary distribution is matrix-geometric, $\boldsymbol{\pi}_{n+1} = \boldsymbol{\pi}_n \boldsymbol{R}$, where $\boldsymbol{R}$ solves $\boldsymbol{A}_0 + \boldsymbol{R}\boldsymbol{A}_1 + \boldsymbol{R}^2\boldsymbol{A}_2 = \boldsymbol{0}$ [91]. Networks are handled by decomposition: each station is reduced to a quasi-birth-death queue, and inter-station traffic is propagated as a marked MAP (MMAP), iterating until the flows stabilize. This makes the solver the tool of choice for models whose temporal correlations (autocorrelated arrivals, non-renewal departures) invalidate product-form and two-moment decompositions.

The solver resides in `jline/solvers/mam/`, with QBD, fitting, and process-algebra algorithms in `jline/api/mam/` and `jline/api/map/`, several of which are ports of the BuTools [62] and SMCSolver/Q-MAM [10] libraries.

7.2 Analyzers and methods

`Solver_mam_analyzer` first recognizes closed-form cases: an isolated MAP/MAP/1 station routes to `Solver_mam_mapmap1_exact`, and an M/M/c($/K$) station to `Solver_mam_mmck_exact`. Otherwise it dispatches by method:

`default`, `dec.mmap` The MMAP decomposition pipeline (`Solver_mam_basic`, `Solver_mam_basic_mmap`): each station is analyzed as a MAP/MAP/c QBD, its departure process is approximated as an MMAP preserving marginal rates and lag correlations, and `Solver_mam_traffic` implements the flow operations (superposition, Bernoulli splitting, class marking) in the MMAP algebra [17,61]. The iteration continues until arrival-process parameters converge.

`dec.source`, `dec.poisson` Simplified decompositions in which internal flows are replaced by renewal (source-equivalent) or Poisson processes, providing cheaper and more robust, if less accurate, baselines; `Qna_superpos` supplies the two-moment superposition used in hybrid schemes [72, 120].

mna The Matrix Network Analyzer (`Solver_mna`, `Solver_mna_open`, `Solver_mna_closed`), a decomposition algorithm for phase-type queueing networks that propagates full Markovian representations of the flows and supports both open and closed topologies [80, 81].

inap An iterative network approximation (`Solver_mam_inap`) that alternates per-station QBD solutions with global rate balancing until a fixed point; a compositional aggregation variant is available through `Solver_mam_ag` and `Solver_mam_build_ag`, which constructs aggregated background processes in the spirit of RCAT-based product-form composition [83, 84].

exact Direct QBD solution when the whole model maps to a single level-structured chain.

Batch arrivals and services are covered by the BMAP/MAP/1 and MAP/BMAP/1 handlers (`Solver_mam_bmap_map_1`, `Solver_mam_map_bmap_1`) [17,92]. Retrial queues with orbit are solved by `Solver_mam_retrial` and the algorithm `Qsys_bmapphnn_retrial`, using the level-dependent constructions of retrial theory [97]. Fork-join stations use `Solver_mam_fj`. Priority queues are treated through `Priority Analysis` with the ETAQA-style aggregate solutions of Riska and Smirni [100, 101].

7.3 Level-dependent and transient QBDs

Load-dependent stations yield level-dependent QBDs (LDQBDs), solved by `Solver_mam_ldqbd` with the matrix-continued-fraction construction of Bright and Taylor [15]; the `statevec` variant returns the full level-phase distribution for consumption by `SolverENV` (Chapter 10). Transient analysis is provided by `Solver_mam_transient_qbd` and `Solver_mam_ldqbd_transient`, which compute time-dependent level distributions by Laplace-transform methods with numerical inversion, automatically selected by `getTranAvg` for MAP/MAP/1($/N$) stations. First-passage metrics (response-time distributions, `getCdfRespT`) are computed by `Solver_mam_passage_time` from the fundamental-period matrices $\boldsymbol{G}$ and $\boldsymbol{U}$ of the QBD [76].

The QBD algorithms in `jline/api/mam/` implement cyclic reduction and logarithmic reduction for the $\boldsymbol{R}$ and $\boldsymbol{G}$ matrices with the numerically stable variants of [10]; spectral shortcuts are used when the phase spaces are small. MAP fitting utilities in `jline/api/map/` include moment and autocorrelation fitting, notably the KPC-Toolbox recipes [29], two-phase MMPP fitting [60, 61], and the count-process moment matching used by trace-driven workflows [17].

7.4 Architectural flow

Figure 7.1 shows the decomposition path; exact single-station cases bypass the traffic iteration entirely.

7.5 Implementation notes

Phase-space growth is the main cost driver: MMAP superposition multiplies phase dimensions, so the traffic handlers compress departure processes back to small-order MMAPs by moment and correlation matching before propagation, an accuracy-cost trade-off that dominates the method's behavior on large topologies.

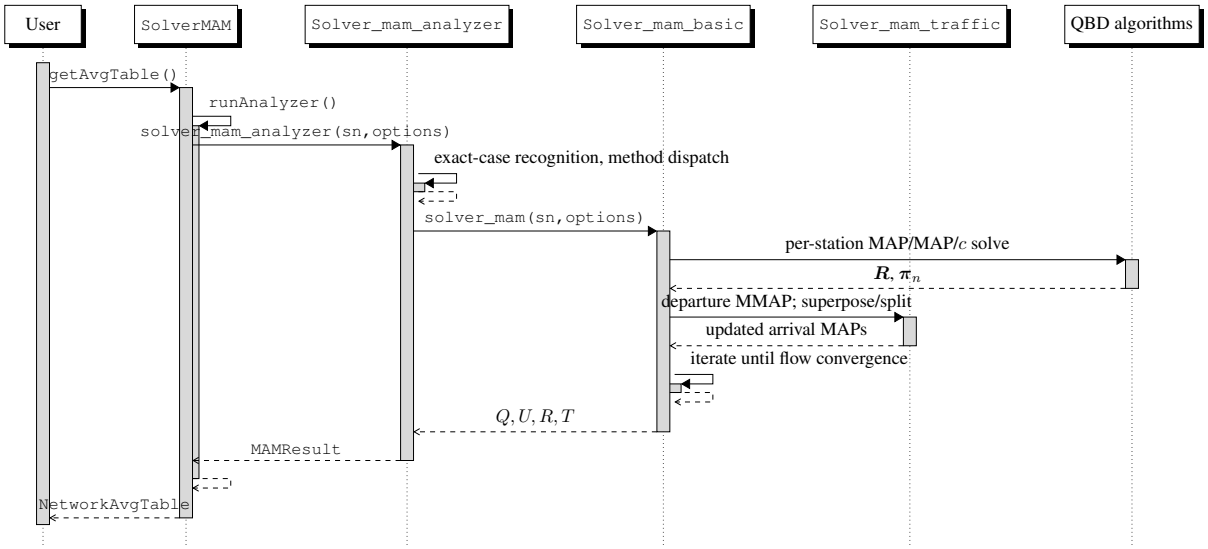

Figure 7.1: Sequence diagram of `SolverMAM` with the MMAP decomposition method.

Default orders and compression tolerances live in `MAMOptions`. Closed networks impose population constraints handled by `Solver_mna_closed` through a population-coupling fixed point. Known accuracy caveats are recorded in the test suite; notably, service-process autocorrelation propagation through multi-hop routes is validated against JMT simulation.

7.6 Feature and method support

Table 7.1 maps modelling features to the `SolverMAM` methods. Per the method-aware gate (`supportsModelMethod`), `mna` rejects mixed open/closed models and `ldqbd` requires a single-class model; the `retrial` and `fj` columns are the retrial-orbit and fork-join analyzers, selected by topology (a fork-join model is routed to `fj`, not rejected). `dec.mmap` (default) propagates full marked-MAP flows; `dec.source` replaces internal flows by renewal approximations, so it captures Markovian arrivals only approximately. General distributions enter through MAP fitting (○).

Table 7.1: Feature support of `SolverMAM` methods (● supported, ○ approximate/conditional, blank unsupported).

Feature	dec.mmap	dec.source	mna	inap	ldqbd	exact	retrial	fj
Job classes								
Open class	●	●	●	●	○	●	●	●
Closed class	●	●	●	●	●			●
Mixed open/closed	●	●		○				○

continued on next page

Feature	dec.mmap	dec.source	mna	inap	ldqbd	exact	retrial	fj
Single-class only					●			
Class switching	●	●	●	●				
Node types								
Source, Sink	●	●	●	●	●	●	●	●
Queueing station	●	●	●	●	●	●	●	●
Delay (infinite server)	●	●	●	●		●		●
Multiserver (MAP/MAP/c)	●	●	●	●		●		
Fork / Join								●
Service and arrival distributions								
Exponential	●	●	●	●	●	●	●	●
Phase-type (Erlang, HyperExp, Coxian, APH, PH)	●	●	●	●	●	●	●	●
MAP, MMPP2, DMAP	●	○	●	●	●	●	●	○
Batch (BMAP, MMAP)	●		○	○		●		
Matrix-exponential (ME), RAP	●	○	○	○		●		
General (Gamma, Weibull, Lognormal, Pareto, Uniform, Det)	○	○	○	○	○	○		
Scheduling disciplines								
INF (delay)	●	●	●	●		●		●
FCFS	●	●	●	●	●	●	●	●
PS	○	○	○	○		○		
HOL (priority)	○					○		
Advanced features								
Retrial (orbit)							●	
Load dependence (LDQBD)					●			
Analysis and output								
Steady-state means (`getAvg`)	●	●	●	●	●	●	●	●
Response-time CDF (`getCdfRespT`)	○				○	●		
Transient averages (`getTranAvg`)	○				○	●		

7.7 Method algorithms

Table 7.2 gives high-level pseudocode for each `SolverMAM` method.

Table 7.2: High-level pseudocode of `SolverMAM` methods.

Method	Pseudocode
`dec.mmap`	1. for each station build the QBD blocks $(\boldsymbol{A}_0, \boldsymbol{A}_1, \boldsymbol{A}_2)$ from arrival and service MAPs 2. solve $\boldsymbol{A}_0 + \boldsymbol{R}\boldsymbol{A}_1 + \boldsymbol{R}^2\boldsymbol{A}_2 = \boldsymbol{0}$ (cyclic/logarithmic reduction); $\boldsymbol{\pi}_{n+1} = \boldsymbol{\pi}_n \boldsymbol{R}$ 3. approximate the departure process as an MMAP; superpose/split/mark into downstream arrivals (compress order) 4. iterate steps 1–3 until the arrival descriptors converge
`dec.source`	as `dec.mmap` but replace internal flows by renewal (source-equivalent) or Poisson processes; cheaper, more robust, less accurate
`mna`	propagate full PH/MAP flow descriptors through the network (open or closed topology); closed models add a population-coupling fixed point
`inap`	alternate per-station QBD solutions with global rate balancing until a network fixed point
`ldqbd`	level-dependent QBD via the matrix-continued-fraction construction (Bright–Taylor); single class; returns the level-phase distribution

continued on next page

Method	Pseudocode
`exact`	map the whole model to one level-structured chain and solve its $\boldsymbol{R}/\boldsymbol{G}$ matrices directly
`retrial`	build the level-dependent QBD with an orbit level; solve the matrix-geometric orbit/queue distribution
`fj`	model the fork-join synchronization as a QBD over sibling-task states and solve the join completion

7.8 Bibliographic notes

Matrix-geometric methods originate with Neuts [91,92]; algorithmic treatments and software are covered by Latouche and Ramaswami [76] and by the SMCSolver/Q-MAM tools [10], and the MAMSolver line [100, 101]. LDQBD algorithms are due to Bright and Taylor [15]. MAP/MMAP modeling and fitting are surveyed in [17], with fitting recipes in [29,60,61]. Decomposition of non-renewal networks descends from QNA [72, 120]; the MNA algorithm is introduced in [80, 81]. Retrial queue theory is surveyed in [97]; entropy-based alternatives for general networks appear in [70].

Chapter 8

SolverLDES

8.1 Overview

SolverLDES is the LINE discrete-event simulator, an event-driven simulation engine built on the SSJ stochastic simulation library [78]. In contrast with SolverSSA, which samples the Markov jump process of the phase-type expanded state vector, LDES simulates job entities flowing through the network: arrivals, service completions, and routing decisions are scheduled as timestamped events on a future-event list, and service times are drawn directly from the configured distributions. This makes the solver applicable to non-Markovian timing (deterministic, uniform, Pareto, Weibull, lognormal, empirical traces via Replayer) without phase-type expansion, and to fine-grained scheduling semantics (SRPT, FB, EDF, pass-and-swap, polling, fair queueing variants) that are cumbersome to encode as Markovian synchronizations.

The solver resides in jline/solvers/ldes/, with the simulation algorithm in handlers/Solver_ssj.java and its layered-network counterpart in handlers/Solver_ssj_ln.java. The same JAR backend is invoked by the MATLAB and Python codebases through a subprocess (or an ahead-of-time compiled native binary), so LDES results are bit-identical across codebases for fixed seed and samples. The authoritative statement of supported model features is SolverLDES.getFeatureSet(), and getLNFeatureSet() for layered networks.

8.2 Simulation algorithm

Solver_ssj compiles sn into an array of station runtime objects, each holding its buffer structures, server tokens, and per-class distribution samplers backed by SSJ random streams. The event loop is driven by SSJ's simevents scheduler: simulator.simulate(maxEvents) advances simulated time until options.samples events have been processed (steady-state mode) or until the end of the requested horizon (transient mode, solver_ssj_transient). The main event types are arrivals (from Source nodes or upstream departures), service completions (per in-service job, with preemption bookkeeping for the preemptive-resume and preemptive-repeat disciplines), impatience firings (reneging, balking, retrial orbit re-attempts), cache lookups with replacement-list updates, Petri net transition firings with atomic token consumption and production, and state-change events for modulated arrival processes.

Scheduling is implemented per station by policy-specific insertion and selection rules over the buffer: insertion order realizes FCFS/LCFS/SIRO/priority/deadline orders; size-based policies (SJF, SEPT, SRPT, PSJF, FB/SETF) maintain attained-service or remaining-work keys [93, 121]; processor-sharing families (PS, DPS, GPS) are simulated by rescheduling the earliest completion whenever the sharing weights change; polling stations implement exhaustive, gated, and decrementing service with switchover times [113]; pass-and-swap (PAS) stations apply the swap-graph semantics at each completion. Fork nodes replicate jobs into sibling tasks and join nodes synchronize them, including quorum joins; finite-capacity stations and regions apply blocking-after-service or drop rules at admission.

8.3 Analyzers, estimation, and rewards

`Solver_ldes_analyzer` validates the model against the feature set, normalizes the initial state (closed-class jobs are injected consistently with `sn.state` rather than lumped at the reference station), draws the master random stream from `options.seed`, and calls the algorithm in steady-state or transient mode. `Solver_ldes_analyzer_parallel` runs independent replications on separate SSJ substreams and pools the estimators. Point estimates are computed online: utilizations and queue lengths by time-weighted integrals, throughputs by event counting, and response times by per-job timestamps; transient mode emits trajectories on the requested grid. Reward metrics (`getAvgReward`, `getTranReward`) are computed by event-level integration of user-defined state functionals along the sample path, permitting nonlinear rewards such as $\mathbb{E}[n^2]$, and are validated against the CTMC reward engine. Verification of the engine itself follows a two-sample testing methodology: the acceptance suite compares LDES against `SolverJMT` with two-sample t-tests at the 1% significance level.

Layered networks are simulated natively by `Solver_ssj_ln` through `Solver_ldes_ln_analyzer`: hosts, tasks, entries, activities, and synchronous or asynchronous calls are simulated directly on the LQN topology (Chapter 11 describes the LQN formalism), avoiding the layer-decomposition error of analytical LQN solvers [45].

8.4 Architectural flow

Figure 8.1 shows the standard path; in the MATLAB and Python codebases, the call into `Solver_ldes_analyzer` is preceded by a subprocess dispatch that serializes the model, invokes the JAR or the native binary, and parses the returned result (`LDESResultIO`).

8.5 Implementation notes

The algorithm is a single translation unit by design, keeping the event-handling code paths monomorphic for JIT friendliness; a lightweight standalone artifact (`ldes.jar`, plus a GraalVM native binary) repackages it for tool-independent distribution. Random streams follow SSJ's stream/substream discipline [78], guaranteeing reproducibility and independence across replications. Petri net firings are atomic: input tokens are consumed and output tokens produced in one event, which avoids the artifact states of token-reservation implementations. Cache simulation updates replacement lists (LRU/FIFO/RR/SFIFO) at lookup time and

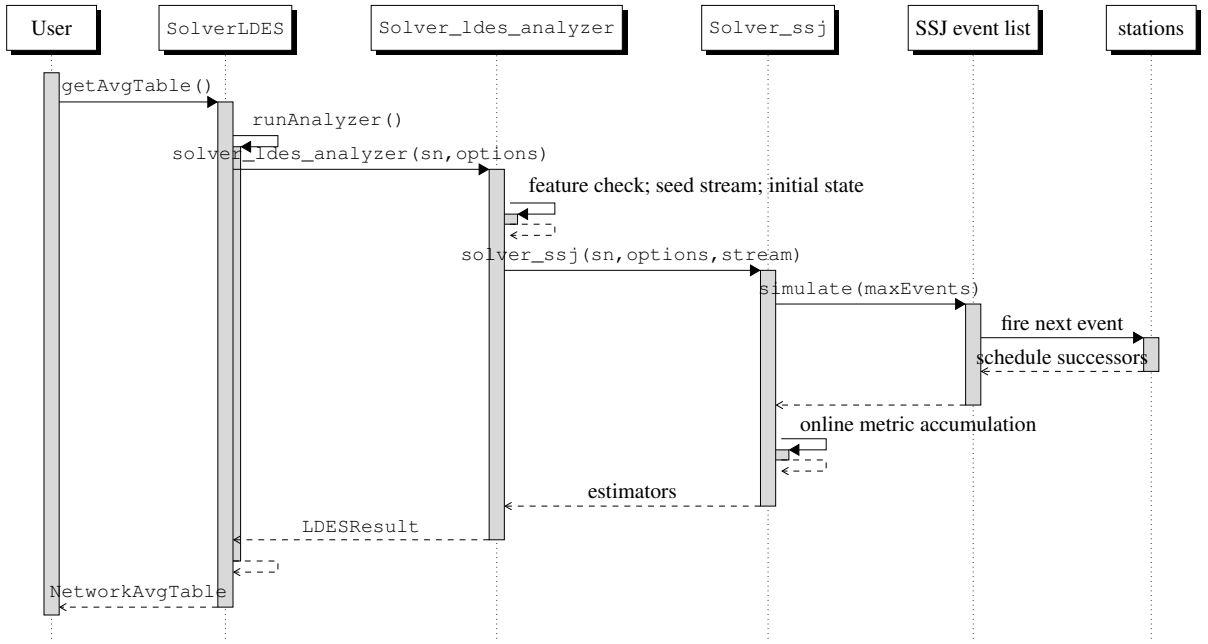

Figure 8.1: Sequence diagram of `SolverLDES` for a steady-state request.

supports delayed-hit retrieval semantics in which concurrent misses on an item coalesce onto one back-end fetch. For LQN models, an external cross-check against the `lqsim` simulator of the LQNS distribution [46] is available through the `SolverLQNS` wrapper.

8.6 Feature and method support

`SolverLDES` has the widest feature envelope in LINE, since discrete-event simulation of job entities imposes no product-form or Markovian restriction. Feature admission does not vary across the simulation methods: the steady-state and transient engines and the parallel-replication engine accept the same models (Table 8.1) and differ only in output (transient trajectories, replication confidence intervals, reward functionals) and cost. The authoritative list is `SolverLDES.getFeatureSet()` (and `getLNFeatureSet()` for layered networks).

Table 8.1: Feature envelope of `SolverLDES` (• supported).

Feature	LDES	**Feature**	LDES
Job classes and nodes			
Open / closed / self-looping class	•	Class switching	•
Priority classes	•	Source, Sink, Queue, Delay	•
Router	•	Fork / Join (with quorum)	•
Logger / LogTunnel	•	Cache node	•
Place, Transition (timed/immediate)	•	Queueing place (QPN)	•

continued on next page

Feature	LDES	Feature	LDES
Finite-capacity region (FCR)	•	Finite-capacity blocking	•
Service and arrival distributions			
Exponential, Erlang, HyperExp	•	Coxian, Cox2, APH, PH	•
MAP, DMAP, MMAP, BMAP, MMPP2	•	ME, RAP	•
Deterministic, Immediate, Disabled	•	Uniform, Gamma, Pareto	•
Weibull, Lognormal	•	CyclicPoisson	•
Replayer / Trace (empirical)	•		
Scheduling disciplines			
INF, FCFS (+PR/PI)	•	LCFS (+PR/PI)	•
PS, DPS, GPS	•	LPS (limited PS)	•
SIRO, HOL	•	SJF, LJF	•
SEPT, LEPT	•	SRPT (+PRIO)	•
PSJF, FB, SETF, LRPT, FSP	•	EDD, EDF (deadline)	•
PAS (pass-and-swap)	•	POLLING	•
EXT (external self-loop)	•	Priority variants (`*PRIO`)	•
Routing, cache, advanced			
Routing PROB, RAND	•	RROBIN, WRROBIN	•
JSQ, KCHOICES	•	Cache repl. (RR/FIFO/SFIFO/LRU)	•
Delayed-hit retrieval	•	Load dependence	•
Retrial (orbit)	•	Balking	•
Reneging (impatience)	•	Multiserver ($c > 1$)	•
Layered networks and output			
LQN (host/task/entry/activity)	•	Sync / async calls	•
Activity precedence (SEQ/AND/OR)	•	Steady means (`getAvg`)	•
Transient (`getTranAvg`)	•	Rewards (`getAvgReward/getTranReward`)	•

8.7 Method algorithms

Table 8.2 gives high-level pseudocode for the simulation engine and its variants.

Table 8.2: High-level pseudocode of `SolverLDES` engines.

Engine	Pseudocode
steady-state	1. compile `sn` to station runtime objects with per-class SSJ samplers; seed streams from `seed`; inject the initial state 2. pop the next event from the SSJ future-event list (arrival, completion, routing, impatience, cache lookup, transition firing, modulation) 3. apply it to the station and schedule successor events; update time-weighted queue/utilization integrals, event counts, and per-job timestamps 4. repeat until `samples` events; remove warm-up; report Q, U, R, T
transient	as above but advance to the requested horizon and emit trajectories on the time grid
`parallel`	run independent replications on separate substreams; pool the estimators and report confidence intervals
reward	integrate user-defined state functionals along the sample path (`getAvgReward/getTranReward`)
LQN (`ssj_ln`)	simulate hosts, tasks, entries, activities, and synchronous/asynchronous calls directly on the LQN topology

8.8 Bibliographic notes

The SSJ library and its stream architecture are described by L'Ecuyer et al. [78]. Scheduling-policy analysis relevant to the implemented disciplines is surveyed in [93, 121], polling systems in [113], and MAP/trace-driven input modeling in [17, 29]. The simulator's role within LINE, including its validation methodology against JMT, is discussed in [22].

Part II

Meta-solvers

Chapter 9

SolverAUTO

9.1 Overview

`SolverAUTO` is the automatic solver-selection meta-solver: given a model, it chooses which concrete solver and method to run. Solver selection is a nontrivial decision problem because the accuracy and cost of each paradigm depend on structural model features (product-form compliance, population size, distribution classes, scheduling disciplines) in ways that are only partially characterized analytically; `AUTO` encodes this decision as an explicit, inspectable policy. The class resides in `jline/solvers/auto/`, together with the `LINE` convenience facade and a `ModelAnalyzer` component that extracts decision features from `sn`.

9.2 Selection policy

The default policy is rule-based. `ModelAnalyzer` computes a feature vector including: openness of the workload (open, closed, mixed), number of stations, classes, and jobs, presence of non-exponential distributions and their SCVs, scheduling disciplines, load dependence, and topology properties (tandem, cyclic, fork-join, caches, Petri net elements). The heuristic then proceeds by elimination and preference:

1. candidate solvers are instantiated (`SolverMVA`, `SolverNC`, `SolverMAM`, `SolverFluid`, and the simulators) and filtered by `supports(model)`, i.e., feature-set admission;

2. among the admissible candidates, analytical solvers are ranked by expected accuracy on the detected structure: exact paradigms first when their cost bound is acceptable (exact MVA or NC for small product-form models), then the approximation whose known error profile best matches the model (AMVA for large closed product-form networks, MAM for autocorrelated open traffic, Fluid for very large populations);

3. simulation (`JMT`, `LDES`, `SSA`) is the fallback when no analytical candidate admits the model or when the request targets metrics unavailable analytically.

The choice is also sensitive to the invoked handle: `getAvg` requests may be answered by a different solver than `getTranAvg` or `getCdfRespT` on the same model, since transient and distributional support differs

across solvers. The selected solver's name is recorded in the result for reproducibility. A learning-based selection mode, in which the ranking is produced by a classifier trained on solved instances, is scaffolded in the implementation and falls back to the heuristic; the general approach of learning solver selection and surrogate models of queueing networks is an active research direction [48].

9.3 Architectural flow

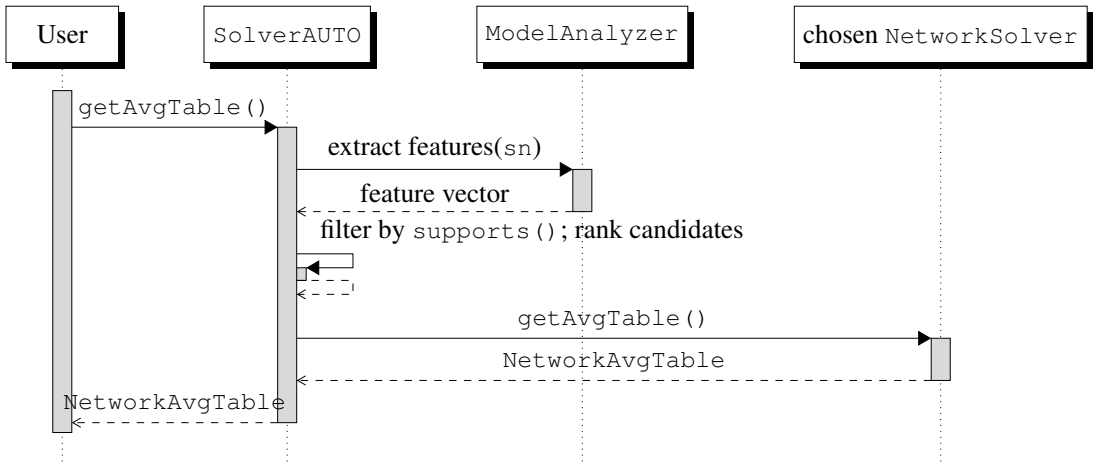

Figure 9.1: Sequence diagram of `SolverAUTO`: selection followed by delegation.

Figure 9.1 shows the delegation structure: `AUTO` performs no numerical work itself and inherits the full result object, including solver-specific diagnostics, from the delegate.

9.4 Feature and method support

`SolverAUTO` performs no numerical work of its own: the admissible features are the union of those of the candidate solvers (Chapters 2–8), and support for any given model reduces to that of the delegate it selects. What is specific to `AUTO` is the mapping from detected model structure to the selected solver, summarized in Table 9.1; the exact per-method support then follows from the corresponding chapter table.

Table 9.1: Structure-to-solver selection policy of `SolverAUTO` (default handle `getAvg`).

Detected model structure	Selected solver (method)
Small product-form closed/mixed	NC/MVA (exact)
Large closed product-form	MVA (amva)
Very large populations	`Fluid` (closing)
Open with autocorrelated / MAP traffic	MAM (dec.mmap)
Open G/G/k decomposable	MVA (qna)
Cache network	MVA/NC (cache analyzer)

Detected model structure	Selected solver (method)
Small non-product-form state space	`CTMC`
Stochastic Petri net	`CTMC` / `LDES`
Layered network (LQN)	`LN`
No admissible analytical solver	`LDES` / `SSA` / `JMT` (simulation)
Transient / response-time distribution	`Fluid` / `CTMC` / `MAM`

9.5 Method algorithms

Table 9.2: High-level pseudocode of `SolverAUTO`.

Method	Pseudocode
heuristic (default)	1. extract the feature vector from `sn` (`ModelAnalyzer`): openness, sizes, distributions/SCVs, disciplines, load dependence, topology 2. instantiate candidate solvers and filter by `supports(model)` 3. rank the admissible candidates by expected accuracy vs. cost for the detected structure and the invoked handle (`getAvg`/`getTranAvg`/`getCdfRespT`) 4. delegate to the top solver; record its name in the result
learned	replace step 3 ranking by a classifier trained on solved instances; fall back to the heuristic

9.6 Bibliographic notes

The layered solver taxonomy that makes automated selection well-posed in LINE is presented in [22]. Method-selection trade-offs among exact, asymptotic, and iterative product-form algorithms are analyzed in [21, 24]; the respective domains of decomposition and simulation methods are discussed in the chapters of Part I. Machine-learned performance models as surrogates for queueing solvers are explored in [48].

Chapter 10

SolverENV

10.1 Overview

`SolverENV` analyzes queueing networks operating in a random environment: a finite set of environment stages $e = 1, \ldots, E$, each associated with a `Network` submodel describing the system dynamics while the environment sojourns in that stage, and a stochastic process governing stage transitions. Typical applications are systems with breakdowns and repairs, bursty reconfigurations, or phased workloads [27, 95]. The environment process is specified by inter-stage transition distributions; Markovian environments yield an exact modulated-chain semantics, while phase-type stage holding times are supported by expansion. The solver is an ensemble meta-solver (`jline/solvers/env/`): the user supplies one `NetworkSolver` per stage, and `SolverENV` coordinates them.

10.2 The blending method

The default analysis implements the blending scheme of Casale, Tribastone and Harrison [28]. Let $\boldsymbol{\pi}^{(e)}$ denote the metric vector (e.g., mean queue lengths) of the stage-e submodel, and let α_e be the stationary probability that the environment resides in stage e, computed from the environment generator or its semi-Markov embedding. Blending accounts for the fact that a stage does not start from its own steady state but from the state inherited at the preceding transition: the method iterates

1. solve each stage submodel for its equilibrium metrics, given its current initialization;

2. compute stage-entry distributions by applying the user-specified reset functions (`resetQLFun`, mapping queue lengths at a transition $e \to h$ to the initial condition of stage h);

3. update the per-stage transient-averaged metrics over the stage holding time, using the stage solver's transient handle;

4. mix the stage metrics with weights α_e and repeat until the blended metrics converge.

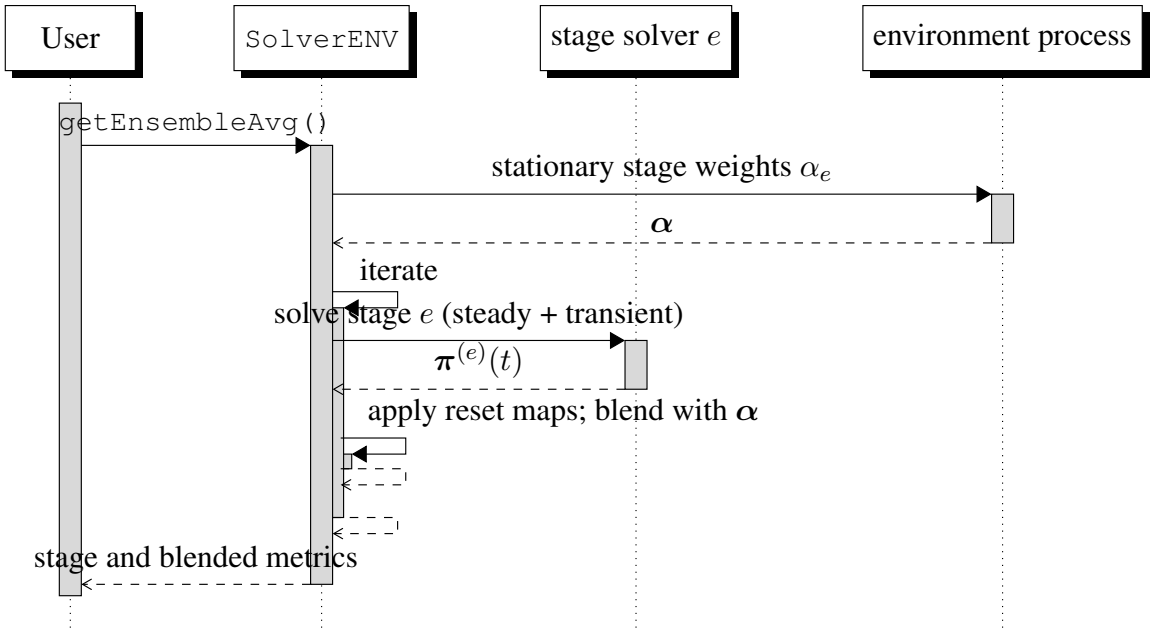

Figure 10.1: Sequence diagram of `SolverENV` with the blending method.

This scheme is exact in the limits of fast and slow environments and interpolates between them; its accuracy for two-stage fluid environments is studied in [27]. Because step 3 requires transient solutions, the natural stage solvers are `SolverFluid`, `SolverCTMC`, and `SolverMAM`.

10.3 The state-vector method

The `statevec` method avoids the blending approximation for stages whose submodels admit an explicit state-space representation. Each stage is compiled to a generator (CTMC backend) or to a level-structured process (MAM/LDQBD backend, Section 7.3); the solver then assembles the modulated joint process over (environment stage, system state), with per-stage entry distributions `piEnter`, per-destination exit distributions `piExitDest`, and sojourn-end distributions `piTimeAvg` computed from the stage holding-time distributions. Metrics follow by direct solution of the joint process. This is exact for Markovian environments and phase-type holding times, at the cost of the product state space; it corresponds to the classical Markov-modulated construction of queues in random environments [91, 105]. Semi-Markov environments with general holding times are supported through the `smp` option by embedding at transition epochs.

10.4 Architectural flow

Figure 10.1 shows the blending loop; the `statevec` method replaces the loop body with the assembly and solution of the joint modulated process.

10.5 Implementation notes

Stage submodels must be structurally compatible (same stations and classes) so that reset maps are well-defined; `SolverENV` verifies stage-solver admission (`supports`) upfront and rejects non-Markovian transition distributions unless the semi-Markov path is selected. The blending iteration reuses each stage solver instance, so expensive structural compilations are performed once per stage. The `statevec` backends are validated to produce bit-identical results across the MATLAB, Java, and Python implementations.

10.6 Feature and method support

`SolverENV` is a meta-solver: the modelling features admissible *within* a stage are those of the per-stage solver, so the columns of Table 10.1 concern the treatment of the environment process and holding times, which is where the methods differ. `blending` requires transient stage solutions (natural backends: Fluid, CTMC, MAM) and is exact in the fast- and slow-environment limits; `statevec` assembles the exact joint (stage, state) process from CTMC or MAM/LDQBD stage backends; `smp` embeds a semi-Markov environment at transition epochs.

Table 10.1: Feature support of `SolverENV` methods (● supported, ○ approximate/conditional, blank unsupported).

Feature	blending	statevec	smp
Environment process			
Markovian environment	●	●	○
Semi-Markov environment	○		●
Phase-type stage holding times	●	●	●
General stage holding times	○		●
Exact for Markovian environment		●	
Stage submodel and coupling			
Open / closed stage classes	●	●	●
Class switching	●	●	●
Reset maps (`resetQLFun`)	●	○	○
Fluid stage backend	●		
CTMC stage backend	●	●	●
MAM / LDQBD stage backend	●	●	○
Output			
Ensemble averages (`getEnsembleAvg`)	●	●	●
Per-stage metrics	●	●	●

10.7 Method algorithms

Table 10.2: High-level pseudocode of `SolverENV` methods.

Method	Pseudocode
`blending`	1. compute stationary stage weights α_e from the environment generator 2. repeat: solve each stage submodel (steady + transient over the stage holding time) from its current initialization 3. apply the reset maps (`resetQLFun`) to obtain stage-entry distributions 4. mix the per-stage metrics with weights α_e 5. until the blended metrics converge
`statevec`	1. compile each stage to a generator (CTMC) or level-structured process (MAM/LDQBD) 2. assemble the joint (stage, system state) modulated process with `piEnter`/`piExitDest`/`piTimeAvg` 3. solve the joint process directly (exact for Markovian environments)
`smp`	embed a semi-Markov environment at transition epochs; weight stage solutions by the embedded stationary distribution and mean holding times

10.8 Bibliographic notes

Blending of randomness in closed networks is introduced in [28]; fluid analysis of two-stage random environments in [27]. Markov-modulated queues are classical in the matrix-analytic literature [76, 91, 105]. The application of LINE to reliability-aware software performance, which motivated this solver, is described in [95], with related model-driven reliability formalisms in [104, 115].

Chapter 11

SolverLN

11.1 Overview

`SolverLN` is the native layered-network meta-solver of LINE. It evaluates layered queueing networks (LQNs), the software-performance formalism in which processors host tasks, tasks expose entries, entries execute activity graphs, and activities issue synchronous or asynchronous calls to other entries [44, 45, 102]. Contention arises simultaneously at hardware (processor) and software (task multiplicity) layers, and the standard solution strategy is layer decomposition: the LQN is split into an ensemble of ordinary queueing networks, one per layer, whose parameters depend on each other's solutions; a fixed-point iteration alternates layer solutions until the metrics converge. This is the architecture of the classical LQNS solver's method of layers [102] and of SRVN-style task decomposition [45, 47]; `SolverLN` generalizes it by allowing an arbitrary LINE solver per layer, so each submodel may be solved by MVA, NC, MAM, Fluid, or simulation.

The solver resides in `jline/solvers/ln/` as a subclass of `EnsembleSolver`, operating on the `LayeredNetworkStruct` representation (`lqn`) documented in the user manuals.

11.2 Layer construction

`buildLayers()` traverses the LQN graph and materializes one `Network` submodel per served element: a host layer for each processor (its tasks act as customers competing for the processor) and a task layer for each non-reference task (its callers act as customers competing for the task's entries). `buildLayersRecursive` walks the call graph so that each layer receives: a delay station representing the callers' think and non-local service times, a queueing station representing the served resource with its multiplicity and scheduling discipline, and one closed class per caller chain, with populations equal to caller multiplicities and demands compiled from activity host demands and call means. Phase-two activities, activity precedences (sequences, AND/OR forks and joins, loops), and replies are compiled into the class routing and the service processes of the layer submodels. Caches appear as `CacheTask` elements and produce cache nodes in the affected layers.

11.3 The fixed-point iteration

`runAnalyzer()` executes the ensemble loop, per iteration `it`:

1. `updateLayers(it)`: refresh the parameters of every layer submodel from the metrics of the other layers computed at the previous iteration;
2. solve each layer with its assigned solver (in parallel where the backend permits);
3. `updateMetrics(it)`: map layer results back to LQN elements (entry throughputs and utilizations, activity response times), with a default moment-matching update (`updateMetricsDefault`) and a moment-based variant (`updateMetricsMomentBased`) that propagates higher service moments to make the decomposition sensitive to service variability;
4. `updateThinkTimes(it)`: recompute the callers' effective think times in every layer as the response time they experience elsewhere, which is the coupling device of the method of layers [102];
5. `updateRoutingProbabilities(it)`: refresh call routing for OR-forks and multiplicity-dependent branching;
6. test convergence of entry-level metrics against `options.iter_tol`; under-relaxation and, for models with coexisting fixed points, warm-started initialization stabilize the iteration.

Upon convergence, `getEnsembleAvg` returns per-layer tables and the LQN-level summary table (task and entry throughputs, utilizations, response times). Transient ensemble analysis (`getTranAvg`) integrates the layer submodels' transient solutions over the iteration fixed point.

Because each layer is an ordinary `Network`, all extended features of Part I solvers are available inside layers; e.g., using `SolverFluid` per layer yields a scalable LQN solver in the spirit of fluid LQN analysis [94, 115], while `SolverMAM` layers propagate service variability. Correctness is routinely cross-checked against the external LQNS solver and its `lqsim` simulator [45, 46], accessed through the `SolverLQNS` wrapper, and against native LQN simulation in LDES (Chapter 8).

11.4 Architectural flow

Figure 11.1 shows the ensemble loop. The per-layer solver is selected by a `SolverFactory`, defaulting to `SolverMVA`.

11.5 Implementation notes

The solver caches layer structures across iterations and re-parameterizes them in place; the AMVA layers additionally reuse the previous fixed point as a warm start, which both accelerates convergence and pins the iteration to a consistent solution when the AMVA equations admit multiple fixed points. Convergence is declared on the maximum relative change of entry metrics; a small number of extra smoothing iterations guard against oscillation. Second-phase service and asynchronous calls introduce non-renewal effects that

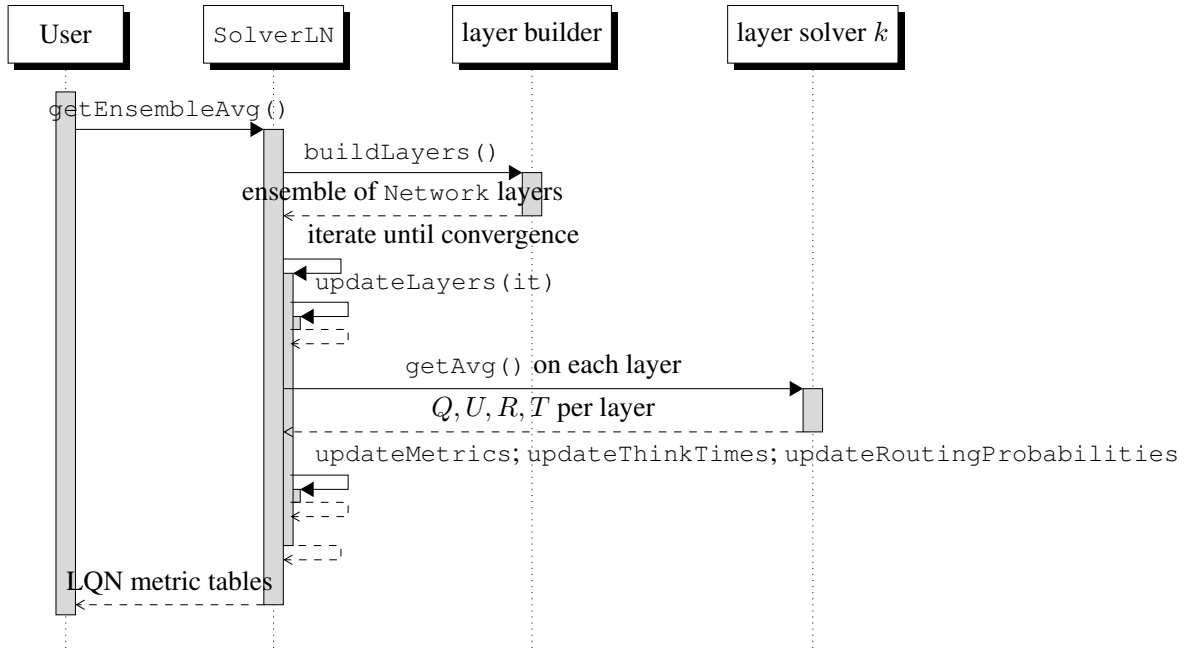

Figure 11.1: Sequence diagram of `SolverLN`: layer decomposition with fixed-point coupling.

layer decomposition captures only approximately; discrepancies of a few percent against simulation on adversarial models are expected and documented in the test suite, consistent with the accuracy envelope reported for analytic LQN solvers [45].

11.6 Feature and method support

`SolverLN` admits the full LQN element vocabulary (`getLNFeatureSet`: hosts/processors, tasks, entries, activities, synchronous and asynchronous calls, and the sequence, AND, and OR activity precedences); every layer is materialized as an ordinary `Network`, so the "method" axis in Table 11.1 is the per-layer solver chosen through the `SolverFactory` (default `SolverMVA`). All per-layer solvers admit the LQN elements; they differ in the within-layer effects they capture, e.g., service-variability sensitivity (moment-based MVA update, MAM, LDES), large-population scaling (Fluid), exact product-form per layer (MVA, NC), and transient ensemble analysis.

Table 11.1: Per-layer solver support in `SolverLN` (● supported, ○ approximate/conditional, blank unsupported).

Feature	MVA (def.)	NC	MAM	Fluid	LDES
LQN elements (admitted by all layer solvers)					

Feature	MVA (def.)	NC	MAM	Fluid	LDES
Host / Processor	●	●	●	●	●
Task (multiplicity)	●	●	●	●	●
Reference task	●	●	●	●	●
Entry, Activity graph	●	●	●	●	●
Synchronous call	●	●	●	●	●
Asynchronous call	●	○	○	○	●
Precedence: sequence	●	●	●	●	●
Precedence: AND fork/join	●	○	○	○	●
Precedence: OR fork/join	●	●	●	●	●
Second phase	●	○	○	○	●
Cache task	●	●	○	○	●
Within-layer capability					
Exact product-form per layer	○	●			
Service-variability sensitivity	○		●	○	●
Large-population scaling	○		○	●	○
Output					
Ensemble averages (`getEnsembleAvg`)	●	●	●	●	●
Transient ensemble (`getTranAvg`)			○	●	●

11.7 Method algorithms

Table 11.2: High-level pseudocode of `SolverLN` (layer decomposition).

Stage / method	Pseudocode
`buildLayers`	traverse the LQN graph; materialize one `Network` per served element (host layer per processor, task layer per non-reference task) with a delay station for callers, a queueing station for the resource, and one closed class per caller chain
iteration	1. `updateLayers(it)`: refresh each layer's parameters from the other layers' previous metrics 2. solve each layer with its assigned per-layer solver (default `SolverMVA`) 3. `updateMetrics`: map layer results back to entries/activities (default or moment-based update) 4. `updateThinkTimes`: set caller think times to the response times experienced elsewhere 5. `updateRoutingProbabilities`: refresh OR-fork and multiplicity-dependent branching 6. test entry-metric convergence vs. `iter_tol` (under-relaxation, warm start); repeat
moment-based	as above but propagate higher service moments in step 3 to make the decomposition sensitive to service variability

11.8 Bibliographic notes

The LQN formalism consolidates the stochastic rendezvous network model of Woodside et al. and the method of layers of Rolia and Sevcik [102]; the canonical solver architecture and semantics are given by

Franks et al. [44–46]. Enhancements relevant to this implementation include activity and phase modeling [47], multiserver approximations for large task pools [123], fluid and mean-field LQN analysis [94,115], and LQN generation from software design models [53, 71, 79]. Applications of `SolverLN` within model-driven engineering pipelines are described in [22, 95].

Chapter 12

SolverUQ

12.1 Overview

`SolverUQ` is the uncertainty-quantification meta-solver. It addresses the setting in which model parameters, typically service or arrival distributions, are not known exactly but only up to a prior: the user attaches a `Prior` object to a service or arrival process, listing alternative distributions with prior weights. Such epistemic uncertainty is the norm when demands are estimated from noisy measurements [110, 118, 119], and propagating it to the performance metrics is preferable to reporting point estimates alone.

The solver resides in `jline/solvers/uq/` as a subclass of `EnsembleSolver`. It is solver-agnostic: a `SolverFactory` supplied at construction decides which concrete solver evaluates each realization.

12.2 Method

`SolverUQ` scans the model for `Prior` occurrences and records, for each, the node index, class index, and whether it parameterizes a service or an arrival process (`PriorInfo`). It then expands the model into the Cartesian family of deterministic networks $\{m_1, \ldots, m_K\}$, one per combination of prior alternatives, with joint weights w_k obtained as products of the marginal prior probabilities. Each realization is evaluated by a solver created from the factory, and the results are aggregated as prior-weighted expectations,

$$\mathbb{E}[\theta] \;=\; \sum_{k=1}^{K} w_k \, \theta(m_k), \tag{12.1}$$

for every metric θ in the average tables; higher posterior functionals (metric variances across realizations, ranges) follow from the same ensemble. This is a discrete-prior Bayesian model-averaging scheme: it is exact with respect to the stated prior, embarrassingly parallel across realizations, and inherits the accuracy profile of the underlying solver. When the priors arise from parameter inference, the weights can encode a posterior computed externally, e.g., by MCMC demand estimation [118] or by maximum-likelihood estimation from queue-length data [119]; surveys of demand-estimation approaches are given in [110].

12.3 Architectural flow

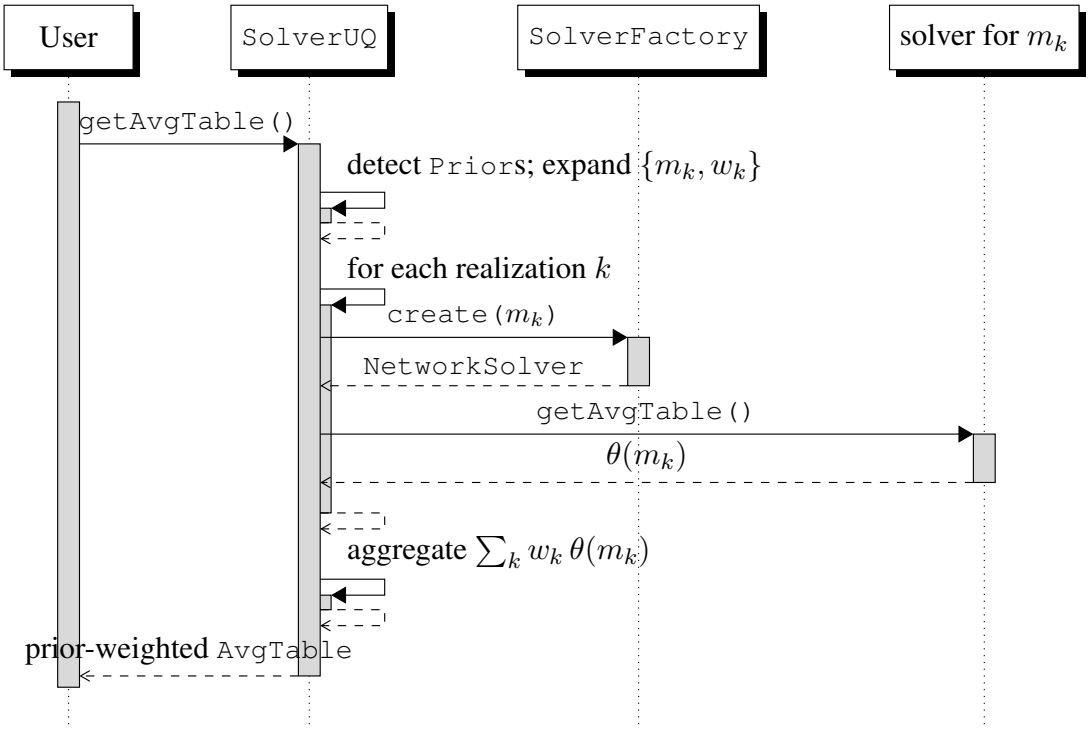

Figure 12.1: Sequence diagram of `SolverUQ`.

Figure 12.1 shows the expansion-evaluation-aggregation pipeline inherited from `EnsembleSolver`.

12.4 Feature and method support

`SolverUQ` is solver-agnostic: each realization m_k is evaluated by a solver built from the supplied `SolverFactory`, so the admissible modelling features are exactly those of that delegate (Chapters 2–8), and every metric the delegate reports is propagated as a prior-weighted expectation (12.1). The only feature specific to `UQ` is the `Prior` attached to a service or arrival process; it is supported wherever the delegate admits the corresponding deterministic distributions, since a prior is expanded into the Cartesian family of deterministic realizations before solution. Consequently `UQ` adds no per-method feature table of its own: consult the table of the factory solver for the concrete envelope, and read every supported metric as its posterior expectation (with variance and range available from the same ensemble).

12.5 Method algorithms

Table 12.1: High-level pseudocode of `SolverUQ`.

Method	Pseudocode
model averaging	1. scan the model for `Prior` occurrences (`PriorInfo`: node, class, service/arrival) 2. expand the Cartesian family $\{m_k\}$ of deterministic realizations with joint weights $w_k = \prod$ marginal prior probabilities 3. evaluate each m_k with a solver from the `SolverFactory` (embarrassingly parallel) 4. aggregate $\mathbb{E}[\theta] = \sum_k w_k\, \theta(m_k)$ per metric; variances and ranges from the same ensemble

12.6 Bibliographic notes

Bayesian inference of queueing model parameters is developed in [118]; likelihood-based demand estimation in [119]; a comparative survey of resource-demand estimation appears in [110] and estimation in multi-threaded applications in [96]. Blending analyses that similarly average network solutions over an exogenous randomization are treated in [28].

Part III

Wrappers

Chapter 13

SolverJMT

13.1 Overview

`SolverJMT` is a wrapper around Java Modelling Tools (JMT) [8], an established open-source suite for the analysis of queueing networks by discrete-event simulation and by analytical product-form algorithms. LINE does not reimplement the JMT algorithms; the wrapper marshals the `sn` structure into a JMT input document, launches the appropriate JMT engine as a subprocess, and parses the results back into the standard LINE result object. The wrapper is the default simulation reference against which the native simulators (`SolverSSA`, `SolverLDES`) are validated, since JMT is an independently developed and widely used tool.

The wrapper resides in `jline/solvers/jmt/`. It requires `JMT.jar` on the classpath; the external GUI need not be installed.

13.2 Wrapper architecture

Two JMT engines are targeted, selected by `options.method`:

JSIMG (simulation) The default path exports `sn` to the JMT simulation format (`jsimg`/`jsimw` XML), including nodes, classes, routing, phase-type service, finite capacity, fork-join, blocking regions, and Petri net primitives. The wrapper writes the seed and the number of samples, runs the JMT simulator, and parses the per-measure output. Because seed and sample budget are marshalled explicitly, JSIMG results are reproducible and match LINE's native simulators for a fixed assignment.

JMVA (analytical) For product-form models the wrapper writes a JMVA document (`writeJMVA`) and sets the `algType` element from the method name: exact MVA (default), RECAL (`jmva.recal`), CoMoM (`jmva.comom`), Chow (`jmva.chow`), Bard-Schweitzer (`jmva.bs`/`jmva.amva`), AQL (`jmva.aql`), Linearizer (`jmva.lin`), and De Souza-Muntz Linearizer (`jmva.dmlin`), with tolerance and `maxSamples` attributes.

The `JMTResult`/`JMTOptions` classes carry the parsed metrics and the wrapper configuration; a temporary working directory holds the exported model and the JMT output, and is cleaned up after parsing.

13.3 Feature and method support

JMT has the broadest feature envelope among the analytical/simulation backends (Table 13.1); admission does not vary between the JSIMG engine and the network structure, whereas the JMVA analytical methods are restricted to product-form models. Non-Markovian distributions are simulated directly by JSIMG.

Table 13.1: Feature envelope of `SolverJMT` (● supported by JSIMG; ○ JMVA product-form only).

Feature	JMT	Feature	JMT
Open / closed / self-looping class	●	Class switching	●
Priority classes	●	Source, Sink, Queue, Delay	●
Router, Logger	●	Fork / Join	●
Cache node	●	Place, Transition, Queueing place	●
Finite-capacity blocking / region	●	Product-form (JMVA)	○
Exponential, Erlang, HyperExp	●	Coxian, Cox2, APH, PH	●
MAP, MMPP2	●	Normal, Gamma, Pareto, Weibull, Lognormal, Uniform, Det	●
Replayer / Trace	●	Cache repl. (RR/FIFO/SFIFO/LRU)	●
INF, FCFS, PS, DPS, GPS	●	LCFS, LCFS-PR	●
SIRO, HOL	●	SEPT, LEPT, SJF, LJF	●
SRPT (+PRIO)	●	LPS, POLLING	●
Priority variants (`*PRIO`)	●	EXT (external self-loop)	●
Routing PROB, RAND	●	RROBIN, WRROBIN	●
JSQ, KCHOICES	●	Steady means (`getAvg`)	●
Response-time distribution	●	Transient / confidence intervals	●

Table 13.2 gives high-level pseudocode of the wrapper dispatch.

Table 13.2: High-level pseudocode of `SolverJMT` methods.

Method	Pseudocode
JSIMG (default)	1. export `sn` to `jsimg` XML (nodes, classes, routing, distributions, capacity, measures) 2. write `seed` and `samples`; run the JMT simulator as a subprocess 3. parse the per-measure results into `JMTResult`
`jmva.*`	1. export `sn` to JMVA XML (`writeJMVA`); set `algType` (MVA/RECAL/CoMoM/Chow/Bard-Schweitzer/AQL/Linearizer/DM-Linearizer) with tolerance and `maxSamples` 2. run the JMT analytical engine; parse Q, U, R, T back

13.4 Architectural flow

13.5 Bibliographic notes

JMT is described by Bertoli, Casale and Serazzi [8]; its role within LINE and the validation methodology (two-sample t-tests of the native simulators against JMT) are discussed in [22]. The JMVA analytical algorithms it exposes are the exact and approximate product-form methods of Part I: MVA [99], RECAL [35], CoMoM [20], and the AMVA family [5, 30, 107, 122].

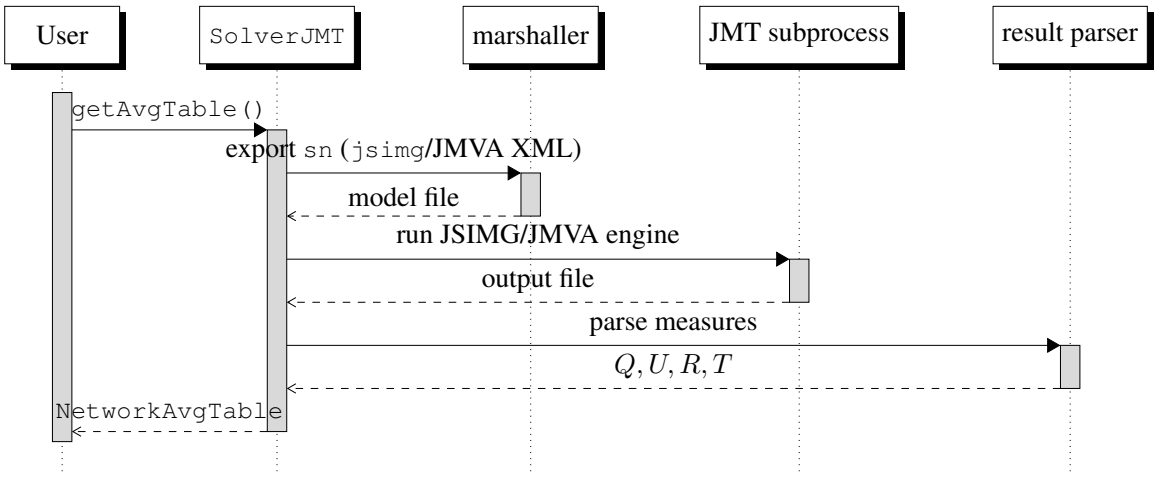

Figure 13.1: Sequence diagram of `SolverJMT`: marshal, run subprocess, parse.

Chapter 14

SolverLQNS

14.1 Overview

`SolverLQNS` is a wrapper around the Layered Queueing Network Solver (LQNS) of Franks et al. [45, 46], the reference tool for the analysis of layered queueing networks by the method of layers and its analytical refinements, together with its discrete-event companion `lqsim`. LINE does not reimplement these algorithms; the wrapper serializes the `LayeredNetwork` model to the LQNS XML interchange, invokes the external `lqns` or `lqsim` command-line tool as a subprocess, and parses the returned results. It is the external cross-check for the native `SolverLN` (Chapter 11) and for native LQN simulation in `SolverLDES`.

The wrapper resides in `jline/solvers/lqns/` and, unlike the network solvers, extends the abstract `Solver` rather than `NetworkSolver`, since its model is an LQN. It requires the `lqns` and `lqsim` executables on the system `PATH`; optionally they may be dispatched through a container (`options.config.container`).

14.2 Wrapper architecture

The wrapper writes the LQN to the `.lqnx`/`.lqxo` XML schema, runs the external tool, and parses the per-element metrics (task and entry throughputs, utilizations, and service/response times) back into the result object:

`lqns` (default) The analytical solver. The method of layers is controlled through LQNS pragmas exposed by `options.config`, notably the multiserver approximation (`conway`, `rolia`, `zhou`, `suri`, `schmidt`, etc.) and the layering and convergence pragmas. LINE writes these pragmas into the invocation and reads the solution once LQNS converges.

`lqsim` (simulation) When `options.method='lqsim'`, the wrapper invokes the LQNS discrete-event simulator instead, passing the simulation length and seed; this provides a simulation ground truth for the same LQN input.

14.3 Feature and method support

The wrapper admits the LQN element vocabulary of `LayeredNetwork` (hosts/processors, tasks, entries, activities with sequence/AND/OR precedences, synchronous and asynchronous calls) as exported to the LQNS schema; the underlying station model is restricted to the disciplines LQNS itself supports (Table 14.1). The `lqns` and `lqsim` methods share the same input; `lqns` is analytical (subject to layer decomposition error) while `lqsim` is a simulation with statistical error.

Table 14.1: Feature support of `SolverLQNS` (● supported, ○ approximate).

Feature	lqns	Feature	lqsim
Host / Processor	●	Host / Processor	●
Task (multiplicity)	●	Task (multiplicity)	●
Reference task	●	Reference task	●
Entry, Activity graph	●	Entry, Activity graph	●
Synchronous call	●	Synchronous call	●
Asynchronous call	●	Asynchronous call	●
Precedence: sequence	●	Precedence: sequence	●
Precedence: AND fork/join	○	Precedence: AND fork/join	●
Precedence: OR fork/join	●	Precedence: OR fork/join	●
Second phase	●	Second phase	●
Multiserver (conway/rolia/...)	○	Multiserver	●
Exp / Erlang / HyperExp / Coxian	●	Non-exponential service	●
FCFS, PS host scheduling	●	FCFS, PS host scheduling	●
Steady means (`getAvg`)	●	Confidence intervals	●

Table 14.2 gives high-level pseudocode of the wrapper dispatch.

Table 14.2: High-level pseudocode of `SolverLQNS` methods.

Method	Pseudocode
`lqns`	1. serialize the LQN to `.lqnx` XML 2. build the `lqns` invocation with multiserver and layering pragmas from `options.config` 3. run `lqns` (locally or in a container); parse the `.lqxo` results into task/entry/activity metrics
`lqsim`	as `lqns` but invoke the `lqsim` simulator with the requested run length and seed; parse the simulated metrics and confidence intervals

14.4 Architectural flow

14.5 Bibliographic notes

The LQNS solver, its method of layers, activity and phase modeling, and the `lqsim` simulator are described by Franks et al. [44–47]; multiserver approximations relevant to the pragmas are surveyed in [34, 112, 123]. The comparison of `SolverLN` against LQNS and the accuracy envelope of layer decomposition are discussed in [22].

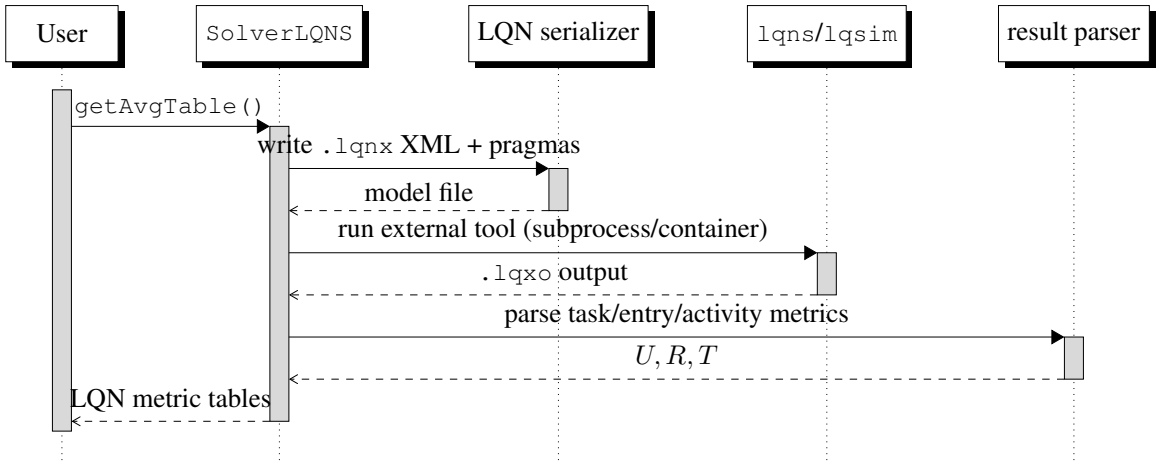

Figure 14.1: Sequence diagram of `SolverLQNS`: serialize LQN, run `lqns`/`lqsim`, parse.

Chapter 15

SolverQNS

15.1 Overview

`SolverQNS` is a wrapper around the external `qnsolver` tool of the RADS/LQNS distribution [45, 46], which analyzes open and closed product-form queueing networks and provides a family of multiserver approximations. LINE marshals the `sn` structure to the `qnsolver` input, runs it as a subprocess, and parses the results. The wrapper is useful for cross-checking the native multiserver AMVA of `SolverMVA` against an independent implementation of the same approximation lineage.

The wrapper resides in `jline/solvers/qns/` with the marshalling in `analyzers/Solver_qns_analyzer`. It requires the `qnsolver` executable to be installed; `Solver_qns_analyzer.isQNSolverAvailable()` guards the invocation.

15.2 Wrapper architecture and methods

`options.method` selects the multiserver approximation, which the wrapper maps to the corresponding `qnsolver` configuration (`options.config.multiserver`): `conway` [34], `rolia` [102], `zhou`, `suri` [112], `reiser` [98], and `schmidt`; `default` leaves the tool's own default in place. The dispatch has two paths:

product-form / open The model is marshalled to `qnsolver` and solved directly (`Solver_qns_analyzer`); the parsed Q, U, R, T are returned.

non-product-form closed (fallback) When the closed model is not product-form, the wrapper converts it to an equivalent `LayeredNetwork` and delegates to `SolverLQNS` (Chapter 14), reusing that wrapper's external solve.

15.3 Feature and method support

Table 15.1 lists the feature envelope; the multiserver methods share it and differ only in the server approximation applied. Non-exponential service is handled to the extent `qnsolver` supports it; correlated (MAP)

input is passed through where the tool admits it.

Table 15.1: Feature envelope of `SolverQNS` (● supported, ○ via LQNS fallback).

Feature	QNS	Feature	QNS
Open class	●	Closed class	●
Self-looping class	○	Class switching	●
Source, Sink, Queue, Delay	●	Router	●
Fork / Join	●	Multiserver ($c > 1$)	●
Non-product-form closed	○	Logger	●
Exponential, Erlang, HyperExp	●	Coxian, Cox2, APH, PH	●
MAP, MMPP2	●	Gamma, Pareto, Weibull, Lognormal, Normal, Uniform, Det	●
Replayer / Trace	●	Place, Transition	●
INF, FCFS, PS, DPS, GPS	●	LCFS, LCFS-PR	●
SIRO, HOL	●	SEPT, LEPT, SJF, LJF	●
EXT (external self-loop)	●	Routing PROB, RAND	●
RROBIN, WRROBIN, KCHOICES	●	Steady means (`getAvg`)	●

Table 15.2 gives high-level pseudocode of the wrapper dispatch.

Table 15.2: High-level pseudocode of `SolverQNS` methods.

Method	Pseudocode
`conway` / `rolia` / `zhou` / `suri` / `reiser` / `schmidt`	1. map the method to the `qnsolver` multiserver configuration 2. if product-form or open: marshal `sn` to `qnsolver`, run the subprocess, parse Q, U, R, T 3. else (non-product-form closed): convert to an equivalent LQN and delegate to `SolverLQNS`

15.4 Architectural flow

15.5 Bibliographic notes

The `qnsolver` tool ships with the LQNS distribution [45, 46]. The multiserver approximations it exposes are those of Conway [34], Rolia and Sevcik [102], Reiser [98], Suri et al. [112], and related schemes [123]; the underlying product-form theory is BCMP [6].

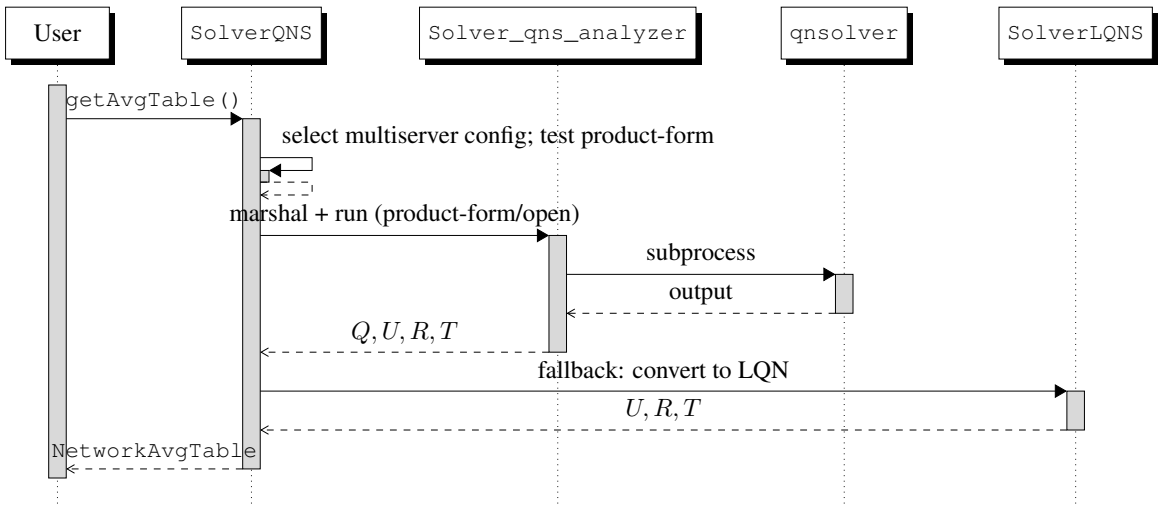

Figure 15.1: Sequence diagram of `SolverQNS`: multiserver dispatch with LQNS fallback.

Part IV

Reference

Appendix A

API Function Reference

This appendix provides a comprehensive catalog of all API functions available in the `jline.api` package. The table lists each function name, its organizational package, and a brief description of its purpose.

Table A.1: Complete API Function Reference

Function Name	Package	Description
`amap2_fit_gamma`	mam	Fits AMAP(2) distributions to match moments and correlation characteristics
`amap2_fit_gamma_map`	mam	Fits AMAP(2) by approximating arbitrary-order MAP with preserved correlation structure
`amap2_fit_gamma_trace`	mam	Fits AMAP(2) from empirical traces while preserving autocorrelation characteristics
`aph2_adjust`	mam	Adjusts moments to ensure feasibility bounds for APH(2) fitting procedures
`aph2_assemble`	mam	Constructs APH(2) transition matrices from specified rates and transition probabilities
`aph2_fit`	mam	Fits APH(2) distributions to match given moments with automatic feasibility adjustment
`aph2_fit_map`	mam	Fits APH(2) distributions by approximating arbitrary-order MAP processes
`aph2_fit_trace`	mam	Fits APH(2) distributions from empirical inter-arrival time traces
`aph2_fitall`	mam	Fits multiple APH(2) distributions to match given moments with exhaustive parameter search
`aph_bernstein`	mam	Constructs APH distributions using Bernstein exponential approximation methods
`aph_convseq`	mam	Convolves a sequence of APH distributions by repeated sequential simplification
`aph_fit`	mam	Fits APH distributions to specified moments using optimization and approximation techniques
`aph_rand`	mam	Generates a random Acyclic Phase-type (APH) distribution with the specified number of phases
`aph_simplify`	mam	Simplifies and combines APH distributions using structural pattern operations
`cache_erec`	cache	Implements exact recursive (EREC) algorithms for cache system analysis
`cache_gamma`	cache	Computes cache access factors from request arrival rates and routing matrices
`cache_gamma_lp`	cache	Computes cache access factors using linear programming optimization methods
`cache_is`	cache	Estimates the cache normalizing constant using Monte Carlo importance sampling

Continued on next page

Table A.1 – API Function Reference. *Continued from previous page*

Function Name	Package	Description
cache_miss	cache	Provides general-purpose algorithms for computing cache miss rates across
cache_miss_asy	cache	Provides asymptotic approximation methods for cache miss rate analysis
cache_miss_fpi	cache	Computes cache miss probabilities using fixed-point iteration methods
cache_miss_is	cache	Computes global, per-user, and per-item cache miss rates using importance sampling
cache_miss_rayint	cache	Estimates cache miss rates using ray method for partial differential equations
cache_miss_spm	cache	Estimates cache miss rates and per-user/per-item miss metrics using the SPM PDE method
cache_mva	cache	Implements Mean Value Analysis algorithms for cache system performance
cache_mva_miss	cache	Implements Mean Value Analysis (MVA) algorithms for computing cache miss
cache_prob_erec	cache	Computes exact cache state probabilities using recursive methods based on
cache_prob_fpi	cache	Computes cache state probabilities using fixed-point iteration algorithms
cache_prob_is	cache	Computes per-item cache hit and miss probabilities using Monte Carlo importance sampling
cache_prob_rayint	cache	Computes cache state probabilities using ray integration methods for
cache_prob_spm	cache	Computes cache state probabilities using the SPM ray-method approximation
cache_rayint	cache	Implements ray integration techniques for cache system analysis using
cache_rrm_meanfield_ode	cache	Implements the mean field ordinary differential equation system for Random
cache_spm	cache	Approximates the cache normalizing constant using the SPM (saddle-point) method
cache_t_hlru	cache	Computes response time metrics for Hierarchical Least Recently Used (H-LRU)
cache_t_lrum	cache	Computes response time metrics for Least Recently Used with Multiple servers
cache_t_lrum_map	cache	Analyzes response times for LRUM cache systems with Markovian Arrival
cache_ttl_hlru	cache	Implements TTL approximation for Hierarchical LRU (H-LRU) cache systems
cache_ttl_lrua	cache	Implementation of Time-To-Live (TTL) approximation for LRU(A) cache systems
cache_ttl_lrum	cache	Implementation of Time-To-Live approximation for Least Recently Used with
cache_ttl_lrum_map	cache	Combines TTL approximation with LRUM cache policies and Markovian Arrival
cache_ttl_tree	cache	Tree-based TTL cache analysis implementation for the LINE solver framework
cache_xi_bvh	cache	Computes cache xi terms using the iterative method from Gast-van Houdt
cache_xi_fp	cache	Estimates cache xi terms using fixed-point algorithms. Xi terms represent
cache_xi_iter	cache	Iteratively computes cache xi terms assuming monotone access factors with list index
ctmc_courtois	mc	Courtois decomposition for nearly completely decomposable CTMCs
ctmc_kms	mc	Koury-McAllister-Stewart aggregation-disaggregation method for CTMCs
ctmc_makeinfgen	mc	Constructs and validates infinitesimal generator matrices for continuous-time
ctmc_multi	mc	Multi-level aggregation method for CTMCs
ctmc_pseudostochcomp	mc	Computes a pseudo-stochastic complement that approximates a CTMC by reweighting eliminated states by stationary probability
ctmc_rand	mc	Generates random infinitesimal generator matrices for continuous-time Markov chains

Continued on next page

Table A.1 – API Function Reference. *Continued from previous page*

Function Name	Package	Description
ctmc_randomization	mc	Converts a continuous-time Markov chain into an equivalent discrete-time chain
ctmc_relsolve	mc	Equilibrium distribution of a continuous-time Markov chain re-normalized with respect to
ctmc_simulate	mc	Generates sample paths for CTMCs using the standard simulation algorithm with
ctmc_solve	mc	Computes the steady-state probability distribution for CTMCs by solving the linear
ctmc_solve_reducible	mc	Solve reducible CTMCs by converting to DTMC via randomization
ctmc_solve_reducible_blkdecomp	mc	Solves reducible CTMCs via SCC-based block decomposition of the generator matrix
ctmc_ssg	mc	Generates the CTMC state space and aggregated representation from a NetworkStruct using the configured cutoffs
ctmc_ssg_reachability	mc	CTMC State Space Generator for Reachability Analysis
ctmc_stmonotone	mc	Computes the stochastically monotone upper bound for a CTMC
ctmc_stochcomp	mc	Implements stochastic complementarity analysis for CTMCs to identify strongly
ctmc_takahashi	mc	Takahashi's aggregation-disaggregation method for CTMCs
ctmc_testpf_kolmogorov	mc	Test if a CTMC has product form using Kolmogorov's criteria
ctmc_timereverse	mc	Computes the infinitesimal generator of the time-reversed continuous-time
ctmc_transient	mc	Computes transient probabilities for CTMCs using numerical integration of the
ctmc_uniformization	mc	CTMC Transient Analysis via Uniformization
dmap_dist	mam	Computes squared L2 distances between joint PMFs and ACFs of two discrete-time MAPs
dmap_moment	mam	Computes raw moments of inter-arrival times of a discrete-time MAP
dmap_pie	mam	Computes the stationary vector at arrival epochs of a discrete-time MAP
dmap_sample	mam	Generates inter-arrival time samples from a discrete-time MAP
dtmc_isfeasible	mc	Check if a matrix represents a feasible DTMC transition matrix
dtmc_makestochastic	mc	Converts non-negative matrices into valid discrete-time Markov chain transition
dtmc_rand	mc	Generates random stochastic transition matrices for discrete-time Markov chains
dtmc_simulate	mc	Generates sample trajectories for DTMCs by sampling from the transition probability
dtmc_solve	mc	Computes the steady-state probability distribution for DTMCs by converting the
dtmc_solve_reducible	mc	Estimates the limiting distribution of a possibly reducible DTMC using strongly connected component decomposition
dtmc_stochcomp	mc	Returns the stochastic complement of a DTMC
dtmc_timereverse	mc	Compute the infinitesimal generator of the time-reversed DTMC
dtmc_uniformization	mc	Computes transient DTMC probabilities by converting to a CTMC generator and applying CTMC uniformization
ldqbd	mam	Computes rate matrices for level-dependent QBD processes via matrix continued fractions
lossn_erlangfp	lossn	Implements fixed-point algorithms for analyzing loss networks using Erlang
lqn_boxbounds	lqn	Computes Majumdar-Woodside robust box bounds on per-task throughput for a layered queueing network (processor-contention model)
lsn_max_multiplicity	lsn	Computes maximum multiplicity constraints for load sharing network (LSN)
m3pp22_fitc_approx_cov	mam.m3pp	Implements parameter fitting for second-order Marked Markov Modulated Poisson Process
m3pp22_fitc_approx_cov_multiclass	mam.m3pp	Implements constrained optimization for fitting M3PP(2,2) parameters given an underlying
m3pp22_interleave_fitc	mam.m3pp	Implements lumped superposition of multiple M3PP(2,2) processes using interleaved

Continued on next page

Table A.1 – API Function Reference. *Continued from previous page*

Function Name	Package	Description
m3pp2m_fitc	mam.m3pp	Implements exact fitting of second-order Marked Markov Modulated Poisson Process
m3pp2m_fitc_approx	mam.m3pp	Implements approximation-based fitting for M3PP(2,m) using optimization methods
m3pp2m_fitc_approx_ag	mam.m3pp	Implements auto-gamma approximation method for M3PP(2,m) parameter fitting
m3pp2m_fitc_approx_ag_multiclass	mam.m3pp	Implements multiclass auto-gamma fitting for M3PP(2,m) with variance and covariance
m3pp2m_fitc_theoretical	mam.m3pp	Fits the theoretical characteristics of an MMAP(n,m) with an M3PP(2,m) under several methods
m3pp2m_fitc_trace	mam.m3pp	Fits an M3PP(2,m) from trace data using counting-process characteristics
m3pp2m_interleave	mam.m3pp	Implements interleaved superposition of multiple M3PP(2,m) processes to construct
m3pp_interleave_fitc	mam.m3pp	Implements fitting and interleaving of k second-order M3PP processes with varying
m3pp_interleave_fitc_theoretical	mam.m3pp	Implements theoretical MMAP fitting through M3PP interleaving using analytical
m3pp_interleave_fitc_trace	mam.m3pp	Implements M3PP interleaving and fitting directly from empirical trace data
m3pp_rand	mam.m3pp	Implements random generation of Markovian Multi-class Point Processes (M3PP)
m3pp_superpos_fitc	mam.m3pp	Implements superposition-based fitting of k second-order M3PP processes into
m3pp_superpos_fitc_theoretical	mam.m3pp	Implements superposition fitting of k second-order M3PP processes using theoretical
m3pp_superpos_fitc_trace	mam.m3pp	Superposes k M3PP processes to fit a multi-class trace with m classes
mamap22_fit_gamma_fs_trace	mam	Fits MAMAP(2,2) from trace data using gamma autocorrelation and forward-sigma characteristics
mamap22_fit_multiclass	mam	Fits MAMAP(2,2) processes for two-class systems with forward moments and sigma characteristics
mamap2m_coefficients	mam	Computes coefficients for MAMAP(2,m) fitting formulas in canonical forms
mamap2m_fit	mam	Fits MAMAP(2,m) processes matching moments, autocorrelation, and class characteristics
mamap2m_fit_fb_multiclass	mam	Fits MAMAP using combined forward and backward moment characteristics for multiclass systems
mamap2m_fit_gamma_fb_mmap	mam	Fits MAMAP with autocorrelation control using forward-backward moments from MMAP input
mamap2m_fit_gamma_fb_trace	mam	Fits a second-order acyclic MMAP from a marked trace matching class probabilities and forward-backward moments
mamap2m_fit_mmap	mam	Fits MAPH/MAMAP(2,m) by approximating characteristics of input MMAP processes
mamap2m_fit_trace	mam	Fits MAMAP(2,m) processes from empirical trace data with inter-arrival times and class labels
map2_fit	mam	Fits MAP(2) processes to match specified moments and autocorrelation decay rates
map2mmpp	mam	Converts MAP representations to MMPP format for compatibility with MMPP-specific algorithms
map2ph	mam	Converts a MAP into a phase-type distribution and the associated PH-renewal process
map_acf	mam	Computes autocorrelation function (ACF) values for MAP inter-arrival times at specified lags
map_acfc	mam	Computes ACFC values for MAP counting processes over time intervals, measuring temporal correlation
map_anfit	mam	Fits a MAP using the Andersen-Nielsen IPP-superposition method calibrated to the Hurst parameter
map_bernstein	mam	Constructs a MAP from a continuous distribution PDF via Bernstein polynomial approximation

Continued on next page

Table A.1 – API Function Reference. *Continued from previous page*

Function Name	Package	Description
map_block	mam	Constructs MAP(2) representations from moment and autocorrelation parameters using fallback
map_ccdf_derivative	mam	Computes derivatives of MAP complementary cumulative distribution functions at zero
map_cdf	mam	Computes CDF values for MAP inter-arrival times using CTMC uniformization techniques
map_checkfeasible	mam	Comprehensive validation of MAP matrices including stochastic properties, numerical stability,
map_count_mean	mam	Computes mean number of arrivals in MAP counting processes over specified time intervals
map_count_moment	mam	Computes power moments of MAP counting processes using moment generating functions and
map_count_var	mam	Computes variance of MAP counting processes over specified time intervals using matrix
map_dist	mam	Computes the squared L2 distance between lag-L joint densities of two MAPs
map_dist_acf	mam	Computes the squared L2 distance between autocorrelation functions of two MAPs
map_embedded	mam	Computes embedded DTMC matrices from MAP representations by extracting transition
map_erlang	mam	Constructs MAP representations of Erlang-k processes with specified means and phases
map_exp_mul_int	mam	Computes the inner product of lag-L joint densities of two MAPs via recursive Sylvester equations
map_exponential	mam	Creates MAP representations of exponential inter-arrival time distributions with specified
map_feasblock	mam	Constructs feasible MAP representations when exact moment matching fails by adjusting
map_feastol	mam	Provides standard tolerance values for numerical feasibility checks in MAP algorithms
map_gamma	mam	Computes gamma parameter measuring autocorrelation decay rates in MAP processes
map_gamma2	mam	Computes largest non-unit eigenvalue of embedded DTMC for MAP correlation characterization
map_hyperexp	mam	Constructs MAP representations of two-phase hyperexponential renewal processes
map_idc	mam	Computes asymptotic index of dispersion for MAP counting processes, measuring long-term
map_infgen	mam	Computes infinitesimal generator matrix of underlying CTMC by combining MAP transition
map_isfeasible	mam	Provides convenient interface for MAP feasibility validation with configurable tolerance
map_joint	mam	Computes joint moments of MAP inter-arrival times for advanced statistical characterization
map_jointpdf_derivative	mam	Computes partial derivatives of MAP joint probability density functions at origin
map_kpc	mam	Computes Kronecker product composition of multiple MAPs for building complex arrival processes
map_kurt	mam	Computes kurtosis of MAP inter-arrival times measuring tail heaviness and distribution shape
map_lambda	mam	Computes the long-run arrival rate of a MAP from its hidden and visible transition matrices
map_largemap	mam	Provides size thresholds for determining when MAP algorithms should switch to
map_mark	mam	Creates Marked MAP (MMAP) representations by adding class labels to MAP arrivals
map_max	mam	Computes MAP representation of maximum inter-arrival times from independent MAP processes

Continued on next page

Table A.1 – API Function Reference. *Continued from previous page*

Function Name	Package	Description
map_mean	mam	Computes the mean inter-arrival time of a MAP as the reciprocal of its arrival rate
map_mixture	mam	Creates probabilistic mixtures of MAP processes with specified mixture probabilities
map_mmpp2	mam	Fits a 2-state MMPP as a MAP from mean, SCV, skewness, and lag-1 autocorrelation
map_moment	mam	Computes raw moments of MAP inter-arrival times using matrix inversion techniques
map_normalize	mam	Sanitizes MAP matrices by ensuring non-negativity constraints and proper diagonal adjustments
map_pdf	mam	Computes PDF values for MAP inter-arrival times using matrix exponential techniques
map_pie	mam	Computes steady-state probability vector of embedded discrete-time Markov chain
map_piq	mam	Computes steady-state probability vector of underlying continuous-time Markov chain
map_pntiter	mam	Computes exact arrival probabilities using iterative numerical methods based on Neuts and Li
map_pntquad	mam	Computes MAP point process probabilities using ODE quadrature methods with Runge-Kutta integration
map_prob	mam	Computes equilibrium probability distribution of underlying CTMC for MAP analysis
map_rand	mam	Generates random MAP representations for testing, simulation, and statistical analysis
map_randn	mam	Generates random MAP samples with added numerical noise for robustness testing
map_renewal	mam	Creates renewal MAP by removing correlations to obtain memoryless arrival processes
map_sample	mam	Generates random samples from MAP distributions for simulation and empirical analysis
map_scale	mam	Rescales MAP inter-arrival time distributions to achieve specified mean values
map_scv	mam	Computes SCV of MAP inter-arrival times as normalized dispersion measure
map_skew	mam	Computes skewness of MAP inter-arrival times measuring asymmetry in distributions
map_stochcomp	mam	Performs state elimination through stochastic complementation while preserving MAP properties
map_sum	mam	Computes MAP representations of sums of identical MAP processes for load scaling
map_sumind	mam	Computes MAP representations of sums of independent MAP processes for modeling
map_super	mam	Creates superposition of MAP processes using Kronecker product techniques
map_timereverse	mam	Computes time-reversed MAP by adjusting transition rates based on stationary distributions
map_var	mam	Computes the variance of MAP inter-arrival times from the second moment and squared mean
map_varcount	mam	Computes variance of event counts in MAP processes over specified time intervals
maph2m_fit	mam	Fits MAPH(2,m) processes to match ordinary moments, class probabilities, and backward moments
maph2m_fit_mmap	mam	Fits MAPH(2,m) by approximating characteristics of input MMAP processes
maph2m_fit_multiclass	mam	Fits MAPH(2,m) models to multiclass characteristics with class-specific parameters
maph2m_fit_trace	mam	Fits MAPH(2,m) from empirical trace data for multiclass service time modeling

Continued on next page

Table A.1 – API Function Reference. *Continued from previous page*

Function Name	Package	Description
mapqn_bnd_lr	mapqn	Implements general linear reduction methods for computing performance
mapqn_bnd_lr_mva	mapqn	Implements linear reduction bounds for MAP queueing networks using Mean
mapqn_bnd_lr_pf	mapqn	Implements linear reduction bounds specialized for product-form MAP
mapqn_bnd_qr	mapqn	Implements general quadratic reduction methods for computing performance
mapqn_bnd_qr_delay	mapqn	Implements quadratic reduction bounds for delay systems in MAP queueing
mapqn_bnd_qr_ld	mapqn	Implements quadratic reduction bounds for load-dependent MAP queueing
mapqn_lpmodel	mapqn	Base class for representing MAP queueing network linear programming models
mapqn_parameters	mapqn	Defines the base parameter structure for MAP queueing network analysis
mapqn_parameters_factory	mapqn	Factory class for creating parameter objects for MAP queueing network
mapqn_qr_bounds_bas	mapqn	Implements Queue-Router bounds using the Balanced Asymptotic Scaling (BAS)
mapqn_qr_bounds_rsrd	mapqn	Implements Queue-Router (QR) bounds using the Randomized Simultaneous
me_mean	mam	Computes the mean of a Matrix Exponential (ME) distribution
me_pie	mam	Computes the stationary initial probability vector for an ME/RAP distribution
me_sample	mam	Generates random samples from an ME distribution using inverse CDF interpolation
me_scv	mam	Computes the squared coefficient of variation of a Matrix Exponential (ME) distribution
me_var	mam	Computes the variance of a Matrix Exponential (ME) distribution
mmap_backward_moment	mam	Computes backward moments of MMAP inter-arrival times for each marked class
mmap_compress	mam	Compresses MMAP using various approximation methods including mixture, matching,
mmap_count_idc	mam	Computes IDC values for each marked class in MMAP counting processes
mmap_count_lambda	mam	Computes arrival rate vectors for each marked class in MMAP processes
mmap_count_mcov	mam	Computes count covariance matrices between marked classes in MMAP processes
mmap_count_mean	mam	Computes mean count vectors for each marked class in MMAP counting processes
mmap_count_var	mam	Computes variance vectors for counting processes of each marked class in MMAP
mmap_cross_moment	mam	Computes cross-moment matrices between different marked classes in MMAP processes
mmap_embedded	mam	Computes embedded discrete-time Markov chain for MMAP processes
mmap_exponential	mam	Constructs MMAP with exponential inter-arrival distributions for each marked class
mmap_forward_moment	mam	Computes forward moments of MMAP inter-arrival times for each marked class
mmap_hide	mam	Hides specified arrival classes in MMAP processes by removing observable events
mmap_idc	mam	Computes asymptotic IDC for each marked class in MMAP as time approaches infinity
mmap_isfeasible	mam	Validates mathematical feasibility of MMAP representations including stochastic
mmap_issym	mam	Checks if an MMAP is symmetric
mmap_lambda	mam	Returns the per-class arrival rate vector of a Marked MAP (alias for mmap_count_lambda)
mmap_maps	mam	Extracts individual MAP processes for each marked class from MMAP representations
mmap_mark	mam	Converts a Markovian Arrival Process with marked arrivals (MMAP) into a new MMAP with redefined classes based on a given probability matrix

Continued on next page

Table A.1 – API Function Reference. *Continued from previous page*

Function Name	Package	Description
mmap_max	mam	Computes element-wise maximum of MMAP processes for synchronization analysis
mmap_mixture	mam	Creates probabilistic mixtures of MMAP processes with specified weights
mmap_mixture_fit	mam	Fits a mixture of Markovian Arrival Processes (MMAPs) to match the given cross-moments
mmap_mixture_fit_mmap	mam	Fits a mixture of Markovian Arrival Processes (MMAPs) to match the given moments
mmap_mixture_fit_trace	mam	Fits an MMAP with m classes from trace data using a mixture of m squared PH-distributions
mmap_mixture_order2	mam	Creates a second-order MMAP mixture from a collection of MMAPs
mmap_modulate	mam	Modulates an MMAP by another MMAP, creating a compound arrival process
mmap_normalize	mam	Normalizes MMAP matrices to ensure feasibility and mathematical validity
mmap_pc	mam	Computes the proportion of counts (PC) for each type in a Markovian Arrival Process with marked arrivals (MMAP)
mmap_pie	mam	Computes steady-state probability vectors for each marked class in MMAP processes
mmap_rand	mam	Generates random MMAP representations for testing and simulation purposes
mmap_sample	mam	Generates random samples from MMAP distributions for each marked class
mmap_scale	mam	Rescales MMAP inter-arrival distributions to achieve specified mean values
mmap_shorten	mam	Converts an MMAP representation from M3A format to BUTools format
mmap_sigma	mam	Computes one-step class transition probabilities for a Marked Markovian Arrival Process (MMAP)
mmap_sigma2	mam	Computes two-step class transition probabilities for a Markovian Arrival Process (MMAP)
mmap_sum	mam	Computes superposition of MMAP processes creating independent multiclass arrival streams
mmap_super	mam	Combines multiple MMAP processes into superposed multiclass arrival streams
mmap_super_safe	mam	Combines multiple MMAPs into a single superposition while keeping the resulting order below a given maximum
mmap_timereverse	mam	Computes the time-reversed version of a Markovian Arrival Process with marked arrivals (MMAP)
mmpp2_fit	mam	Fits MMPP(2) models to match specified moments and correlation characteristics
mmpp2_fit1	mam	Fits MMPP(2) models using simplified single-parameter approach for specific scenarios
mmpp2_fit_count	mam	Fits an MMPP(2) using the Heffes-Lucantoni method based on counting process IDC values
mmpp2_fit_count_approx	mam	Fits an MMPP(2) by optimization-based approximation matching counting-process IDCs
mmpp2_fitc	mam	Fits MMPP(2) models using count statistics and index of dispersion criteria
mmpp2_fitc_approx	mam	Fits MMPP(2) using optimization-based approximation methods for count statistics
mmpp_rand	mam	Generates random MMPP models with diagonal D1 matrices for testing and simulation
ms_additivesymmetricchisquared	measures	Additive symmetric chi-squared distance between two probability distributions
ms_anderson_darling	measures	Implements the Anderson-Darling test for assessing whether a sample comes from
ms_avgl1linfty	measures	Average L1 L-infinity distance between two probability distributions
ms_bhattacharyya	measures	Computes the Bhattacharyya distance measuring the similarity between probability

Continued on next page

Table A.1 – API Function Reference. *Continued from previous page*

Function Name	Package	Description
ms_canberra	measures	Canberra distance between two probability distributions
ms_chebyshev	measures	Chebyshev Distance for Probability Distributions
ms_squaredchisquared	measures	Squared chi-squared distance between two probability distributions
ms_cityblock	measures	Implements the City block distance between probability distributions
ms_clark	measures	Clark distance between two probability distributions
ms_condentropy	measures	Computes conditional entropy measuring the remaining uncertainty
ms_cramer_von_mises	measures	Implements the Cramer-von Mises test statistic for comparing two empirical distributions
ms_cosine	measures	Computes cosine distance (1 - cosine similarity) measuring the angle between two
ms_czekanowski	measures	Czekanowski distance between two probability distributions
ms_dice	measures	Dice distance between two probability distributions
ms_divergence	measures	Divergence distance between two probability distributions
ms_entropy	measures	Implements Shannon entropy for discrete random variables
ms_euclidean	measures	Implements the standard Euclidean distance between probability distributions
ms_fidelity	measures	Fidelity distance between two probability distributions
ms_gower	measures	Gower distance between two probability distributions
ms_harmonicmean	measures	Harmonic mean distance between two probability distributions
ms_hellinger	measures	Computes the Hellinger distance measuring dissimilarity between probability distributions
ms_intersection	measures	Intersection distance between two probability distributions
ms_jaccard	measures	Jaccard distance between two probability distributions
ms_jeffreys	measures	Jeffreys divergence between two probability distributions
ms_jensendifference	measures	Jensen difference divergence between two probability distributions
ms_jensenshannon	measures	Implements Jensen-Shannon divergence, a symmetric and bounded version of
ms_jointentropy	measures	Computes joint entropy measuring the uncertainty of joint distributions
ms_kdivergence	measures	K-divergence between two probability distributions
ms_kolmogorov_smirnov	measures	Implements the Kolmogorov-Smirnov test for determining if a sample follows
ms_kuiper	measures	Implements the Kuiper test statistic, a rotation-invariant variant of the Kolmogorov-Smirnov
ms_kulczynskid	measures	Kulczynski d distance between two probability distributions
ms_kulczynskis	measures	Kulczynski s distance between two probability distributions
ms_kullbackleibler	measures	Implements the Kullback-Leibler divergence measuring distribution differences
ms_kumarhassebrook	measures	Kumar-Hassebrook distance between two probability distributions
ms_kumarjohnson	measures	Kumar-Johnson distance between two probability distributions
ms_lorentzian	measures	Lorentzian distance between two probability distributions
ms_matusita	measures	Matusita distance between two probability distributions
ms_minkowski	measures	Minkowski Distance for Probability Distributions
ms_motyka	measures	Motyka distance between two probability distributions
ms_mutinfo	measures	Computes mutual information measuring the amount of shared information
ms_neymanchisquared	measures	Neyman chi-squared distance between two probability distributions
ms_nmi	measures	Computes normalized mutual information providing a scale-invariant measure
ms_nvi	measures	Computes normalized variation information measuring the normalized distance
ms_pearsonchisquared	measures	Pearson chi-squared distance between two probability distributions
ms_probsymmchisquared	measures	Probabilistic symmetry chi-squared distance between two probability distributions
ms_relatentropy	measures	Computes relative entropy measuring the information difference
ms_product	measures	Product distance between two probability distributions
ms_ruzicka	measures	Ruzicka distance between two probability distributions
ms_soergel	measures	Soergel distance between two probability distributions
ms_sorensen	measures	Sorensen distance between two probability distributions

Continued on next page

Table A.1 – API Function Reference. *Continued from previous page*

Function Name	Package	Description
ms_squaredchord	measures	Squared chord distance between two probability distributions
ms_squaredeuclidean	measures	Squared Euclidean distance between two probability distributions
ms_taneja	measures	Taneja distance between two probability distributions
ms_tanimoto	measures	Tanimoto distance between two probability distributions
ms_topsoe	measures	Topsoe distance between two probability distributions
ms_wasserstein	measures	Implements the Wasserstein distance measuring the minimum cost to transform
ms_wavehegdes	measures	Wave-Hedges distance between two probability distributions
mtrace_backward_moment	trace	Computes backward moments of a multi-class trace
mtrace_bootstrap	trace	Implements bootstrap resampling methods for multi-class empirical trace data
mtrace_count	trace	Computes count statistics from a multi-class trace over specified time windows
mtrace_cov	trace	Computes the covariance matrix for multi-type traces
mtrace_cross_moment	trace	Computes the k-th order moment of the inter-arrival time between an event
mtrace_forward_moment	trace	Computes the forward moments of a marked trace
mtrace_iat2counts	trace	Computes the per-class counting processes of T, i.e., the counts after
mtrace_joint	trace	Given a multi-class trace, computes the empirical class-dependent joint
mtrace_mean	trace	Computes per-class means for multi-class empirical trace data. Enables separate
mtrace_merge	trace	Merges two traces in a single marked (multiclass) trace
mtrace_moment	trace	Computes empirical class-dependent statistical moments for multi-class trace data
mtrace_moment_simple	trace	Computes the k-th order moment of the inter-arrival time between an event
mtrace_pc	trace	Computes the probabilities of arrival for each class
mtrace_sigma	trace	Computes the empirical probability of observing a specific 2-element
mtrace_sigma2	trace	Computes the empirical probability of observing a specific 3-element
mtrace_split	trace	Given a multi-class trace with inter-arrivals T and labels L,
mtrace_summary	trace	Computes summary statistics for multiple trace analysis, providing
npfqn_nonexp_approx	npfqn	Implements approximation methods for non-product-form queueing networks
npfqn_traffic_merge	npfqn	Implements traffic merging algorithms for non-product-form queueing networks
npfqn_traffic_merge_cs	npfqn	Implements traffic merging algorithms for non-product-form queueing networks
npfqn_traffic_split_cs	npfqn	Implements traffic splitting algorithms for non-product-form queueing networks
pfqn_ab	pfqn.ld	Implements the Akyildiz-Bolch linearizer method for analyzing closed product-form queueing networks
pfqn_ab_amva	pfqn.mva	Akyildiz-Bolch AMVA approximation for closed BCMP networks with multi-server stations
pfqn_aql	pfqn.mva	Implements the Aggregate Queue Length (AQL) approximation method for analyzing closed
pfqn_bs	pfqn.mva	Implements the classic Bard-Schweitzer approximate MVA algorithm for closed queueing networks
pfqn_bsfcfs	pfqn.mva	Bard-Schweitzer approximate MVA for FCFS scheduling with optional weighted priorities
pfqn_ca	pfqn.nc	Convolution Algorithm for Product-Form Networks
pfqn_cdfun	pfqn.ld	Provides functionality to evaluate class-dependent scaling functions in load-dependent queueing networks
pfqn_clw	pfqn.nc	Choudhury-Leung-Whitt normalizing constant by numerical inversion of the generating function [32]
pfqn_clw_lld	pfqn.nc	Choudhury-Leung-Whitt inversion extended to limited load-dependent stations via Bertozzi-McKenna transforms [9]
pfqn_comom	pfqn.nc	Computes the log normalizing constant of a multi-station closed network using CoMoM matrix recursion
pfqn_comomrm	pfqn.nc	Implements the Convolution Method of Moments specialized for repairman queueing models

Continued on next page

Table A.1 – API Function Reference. *Continued from previous page*

Function Name	Package	Description
pfqn_comomrm_ld	pfqn.ld	Implements the Convolution Method of Moments (COMOM) for computing normalizing constants in
pfqn_comomrm_ms	pfqn.nc	CoMoM normalizing-constant method for multiserver repairman models with load-dependent rates
pfqn_comomrm_orig	pfqn.nc	Original CoMoM implementation for the single-station finite repairman model
pfqn_conv	pfqn.ld	Multichain convolution algorithm for closed networks with limited joint class dependence (LJCD)
pfqn_conwayms	pfqn.mva	Implements the Conway-Maxwell approximation method for analyzing closed queueing networks
pfqn_cub	pfqn.nc	Implements the cubature (multi-dimensional integration) approach for computing normalizing
pfqn_egflinearizer	pfqn.mva	Implements the Extended General-Form linearizer approximation for closed queueing networks
pfqn_fnc	pfqn.ld	Computes scaling factors for load-dependent functional servers in product-form queueing networks
pfqn_gflinearizer	pfqn.mva	Implements the general-form linearizer approximation for closed queueing networks with
pfqn_gld	pfqn.ld	Implements the generalized convolution algorithm for computing normalizing constants in
pfqn_gld_complex	pfqn.ld	Extends the generalized load-dependent convolution algorithm to handle complex-valued
pfqn_gldsingle	pfqn.ld	Provides specialized auxiliary function for computing normalizing constants in single-class
pfqn_gldsingle_complex	pfqn.ld	Provides specialized auxiliary function for computing normalizing constants in single-class
pfqn_grnmol	pfqn.nc	Computes normalizing constants using the Grundmann-Moeller simplex quadrature rule
pfqn_harel_bounds	pfqn	Computes Harel-Namn-Sturm throughput bounds for closed queueing networks
pfqn_joint	pfqn	Computes joint queue-length probabilities for closed product-form networks via normalizing constants
pfqn_kt	pfqn.nc	Implements the Knessl-Tier asymptotic expansion using the ray method for computing
pfqn_lap	pfqn.nc	Laplace (saddle-point) approximation for the log normalizing constant of a product-form network
pfqn_lcfsqn_ca	pfqn.lcfs	Convolution algorithm for normalizing constants in 2-station LCFS queueing networks
pfqn_lcfsqn_mva	pfqn.lcfs	Exact log-space MVA for 2-station LCFS and LCFS-PR queueing networks
pfqn_lcfsqn_nc	pfqn.lcfs	Computes the normalizing constant of LCFS queueing networks via matrix permanent calculations
pfqn_le	pfqn.nc	Implements the Laguerre expansion approach for computing normalizing constants in
pfqn_le_fpi	pfqn.nc	Implements the fixed-point iteration algorithm used in the Laguerre expansion method
pfqn_le_fpiz	pfqn.nc	Implements the fixed-point iteration algorithm for the Laguerre expansion method
pfqn_le_hessian	pfqn.nc	Computes the Hessian matrix used in the Laguerre expansion method for second-order
pfqn_le_hessianz	pfqn.nc	Computes the Hessian matrix used in the Laguerre expansion method for closed queueing
pfqn_linearizer	pfqn.mva	Linearizer Approximate MVA for Product-Form Networks
pfqn_linearizerms	pfqn.mva	Implements the multi-server version of Krzesinski's linearizer approximation for closed
pfqn_linearizermx	pfqn.mva	Implements linearizer-based approximation methods for mixed queueing networks with

Continued on next page

Table A.1 – API Function Reference. *Continued from previous page*

Function Name	Package	Description
pfqn_linearizerpp	pfqn.mva	Implements the Linearizer++ algorithm for closed queueing networks with enhanced accuracy
pfqn_ljdfun	pfqn.ld	Evaluates limited joint-dependent (LJD) scaling functions via lookup over per-class population vectors
pfqn_lldfun	pfqn.ld	Evaluates limited load-dependent (LLD) scaling functions using spline interpolation for
pfqn_ls	pfqn.nc	Implements the logistic sampling approach for computing normalizing constants in
pfqn_mci	pfqn.nc	Implements Monte Carlo integration approaches including Importance Monte Carlo Integration
pfqn_mmint2	pfqn.nc	Implements numerical integration for computing normalizing constants in multi-class
pfqn_mmint2_gausslaguerre	pfqn.nc	Computes the McKenna-Mitra normalizing constant for repairman models via Gauss-Laguerre quadrature
pfqn_mmint2_gausslegendre	pfqn.nc	Implements Gauss-Legendre quadrature integration for computing normalizing constants
pfqn_mmsample2	pfqn.nc	Implements importance sampling for computing normalizing constants in multi-class
pfqn_mom	pfqn.nc	Implements the Method of Moments using exact arithmetic with BigFraction for computing
pfqn_mu_ms	pfqn.ld	Computes load-dependent scaling factors for multi-server queueing stations with finite
pfqn_mushift	pfqn.ld	Provides utility function for shifting load-dependent scaling vectors by one position,
pfqn_mva	pfqn.mva	Mean Value Analysis for Product-Form Queueing Networks
pfqn_mvald	pfqn.ld	Load-Dependent Mean Value Analysis
pfqn_mvaldms	pfqn.ld	Provides wrapper functionality for load-dependent Mean Value Analysis with automatic
pfqn_mvaldmx	pfqn.ld	Implements Mean Value Analysis for mixed queueing networks with both open and closed classes
pfqn_mvaldmx_ec	pfqn.ld	Provides auxiliary functionality for computing EC terms used in load-dependent Mean Value
pfqn_mvams	pfqn.mva	Provides comprehensive MVA solution for mixed queueing networks with multi-server stations
pfqn_mvamx	pfqn.mva	Implements MVA for mixed networks containing both open and closed classes without multi-server
pfqn_mwrbb	pfqn	Computes Majumdar-Woodside robust box bounds on per-class throughput for closed multiclass networks (NBUE service, FIFO/PS/priority disciplines)
pfqn_nc	pfqn.nc	Normalizing Constant Methods for Product-Form Networks
pfqn_nc_sanitize	pfqn.nc	Sanitizes and preprocesses parameters for product-form queueing network models to
pfqn_nca	pfqn.nc	Implements the Normalizing Constant Approximation method for single-class closed
pfqn_ncld	pfqn.ld	Provides the main entry point for computing normalizing constants in load-dependent
pfqn_nrl	pfqn.nc	Implements the Normal Random Lattice approach for computing normalizing constants
pfqn_nrp	pfqn.nc	Implements the Normal Random Permutation approach for computing normalizing constants
pfqn_panacea	pfqn.nc	Implements the PANACEA approximation method for computing normalizing constants in
pfqn_pff_delay	pfqn.nc	Computes the product-form factor for delay stations in closed queueing networks
pfqn_procomom	pfqn.nc	Computes marginal queue-length probabilities using the Probabilistic CoMoM matrix recursion
pfqn_procomom2	pfqn.ld	Implements the probabilistic class-oriented method of moments for analyzing

Continued on next page

Table A.1 – API Function Reference. *Continued from previous page*

Function Name	Package	Description
pfqn_propfair	pfqn.nc	Implements the proportionally fair allocation method using convex optimization
pfqn_qd	pfqn.mva	Queue-Dependent (QD) approximate MVA solver using gamma and beta scaling functions
pfqn_qzgblow	pfqn.mva	Computes the lower Geometric Bound (GB) for queue lengths in closed single-class queueing
pfqn_qzgbup	pfqn.mva	Computes the upper Geometric Bound (GB) for queue lengths in closed single-class queueing
pfqn_rd	pfqn.nc	Implements the Random Discretization approach for computing normalizing constants in
pfqn_recal	pfqn.nc	Implements the RECAL (Recursive Calculation) algorithm for computing normalizing constants
pfqn_replicas	pfqn	Identifies replicated stations with identical demands and consolidates them into unique stations with multiplicity
pfqn_schmidt	pfqn.ld	Schmidt method for load-dependent MVA with multi-server stations
pfqn_schmidt_amva	pfqn.mva	Schmidt MVA algorithm for multi-class FCFS queueing networks with class-dependent service
pfqn_sqni	pfqn.mva	Implements the Single Queue Network Interpolation method for analyzing multi-class closed
pfqn_stdf	pfqn.nc	Implements McKenna's 1987 method for computing sojourn time distributions at
pfqn_stdf_heur	pfqn.nc	Implements a heuristic variant of McKenna's 1987 method for computing sojourn time
pfqn_xia	pfqn.ld	Implements Xia's asymptotic approximation method for computing normalizing constants
pfqn_xzabalow	pfqn.mva	Computes the lower ABA bound for throughput in closed single-class queueing networks
pfqn_xzabaup	pfqn.mva	Computes the upper ABA bound for throughput in closed single-class queueing networks
pfqn_xzgsblow	pfqn.mva	Computes the lower GSB for throughput in closed single-class queueing networks using
pfqn_xzgsbup	pfqn.mva	Computes the upper GSB for throughput in closed single-class queueing networks using
ph_reindex	mam	Reindexes phase-type distribution maps for network models using integer station and class indices
polling_qsys_1limited	polling	Implements analysis algorithms for 1-limited polling systems where the
polling_qsys_exhaustive	polling	Implements analysis algorithms for exhaustive polling systems where the
polling_qsys_gated	polling	Implements analysis algorithms for gated polling systems where the server
qbd_bmapbmap1	mam	Analyzes batch arrival and service systems using QBD matrix methods
qbd_depproc_etaqa	mam	Constructs a MAP departure-process approximation for MAP/MAP/1-FCFS via ETAQA truncation
qbd_depproc_etaqa_ps	mam	Constructs a MAP departure-process approximation for MAP/MAP/1-PS via ETAQA truncation
qbd_depproc_jointmom	mam	Computes joint moments of consecutive inter-departure times for MAP/MAP/1 queues
qbd_mapmap1	mam	Analyzes MAP/MAP/1 queueing systems using QBD matrix analytic methods
qbd_r	mam	Computes the QBD R matrix using successive substitutions until convergence
qbd_r_logred	mam	Computes QBD R-matrix using logarithmic reduction method for numerical stability
qbd_raprap1	mam	Analyzes RAP/RAP/1 queueing systems using QBD methods with rational arrival processes
qbd_rg	mam	Computes fundamental R and G matrices for QBD analysis of MAP/MAP/1 queues
qbd_setupdelayoff	mam	Analyzes queueing systems with server setup delays and switch-off mechanisms

Continued on next page

Table A.1 – API Function Reference. *Continued from previous page*

Function Name	Package	Description
qsys_gg1	qsys	Provides comprehensive analysis of G/G/1 queues with general arrival and service processes
qsys_gig1_approx_allencunneen	qsys	Implements the widely-used Allen-Cunneen approximation for general G/G/1 queueing
qsys_gig1_approx_gelenbe	qsys	G/G/1 queue approximation using Gelenbe's method
qsys_gig1_approx_heyman	qsys	Analyzes a G/G/1 queueing system using Heyman's approximation
qsys_gig1_approx_kimura	qsys	G/G/1 queue approximation using Kimura's method
qsys_gig1_approx_klb	qsys	Analyzes a G/G/1 queueing system using the Kramer-Langenbach-Belz (KLB) approximation
qsys_gig1_approx_kobayashi	qsys	Analyzes a G/G/1 queueing system using Kobayashi's approximation
qsys_gig1_approx_marchal	qsys	Analyzes a G/G/1 queueing system using Marchal's approximation
qsys_gig1_approx_myskja	qsys	G/G/1 queue approximation using Myskja's method
qsys_gig1_approx_myskja2	qsys	G/G/1 queue approximation using enhanced Myskja's method
qsys_gig1_lbnd	qsys	G/G/1 queue lower bounds
qsys_gig1_ubnd_kingman	qsys	Calculates an upper bound on the waiting time for a G/G/1 system using Kingman's formula
qsys_gigk_approx	qsys	Analyzes a G/G/k queueing system using an approximation method
qsys_gigk_approx_cosmetatos	qsys	G/G/k queue approximation using Cosmetatos method
qsys_gigk_approx_kingman	qsys	Analyzes a G/G/k queueing system using Kingman's approximation
qsys_gigk_approx_whitt	qsys	G/G/k queue approximation using Whitt's method
qsys_gm1	qsys	G/M/1 Queueing System Analysis
qsys_mg1	qsys	Implements the Pollaczek-Khinchine formula for M/G/1 queues with Poisson arrivals
qsys_mg1_prio	qsys	M/G/1 queueing system with non-preemptive class priorities
qsys_mg1k_loss	qsys	M/G/1/K loss probability calculation
qsys_mg1k_loss_mgs	qsys	M/G/1/K loss probability using MacGregor Smith approximation
qsys_mginf	qsys	M/G/inf queue analysis (infinite servers)
qsys_mm1	qsys	Implements exact analytical solutions for the M/M/1 queue (Poisson arrivals, exponential
qsys_mm1k_loss	qsys	M/M/1/K loss probability calculation
qsys_mmk	qsys	Implements exact analytical solutions for M/M/k queues with Poisson arrivals,
qsys_mapg1	qsys	Analyzes MAP/G/1 queues with MAP arrivals and general service times using BUTools
qsys_mapm1	qsys	MAP/M/1 queueing system analysis with MAP arrivals and exponential service
qsys_mapmap1	qsys	Analyzes MAP/MAP/1 queues with MAP arrivals and MAP service times
qsys_mapmc	qsys	MAP/M/c multiserver queueing system analysis with MAP arrivals
qsys_mapph1	qsys	Analyzes MAP/PH/1 queues with MAP arrivals and phase-type service distributions
qsys_phph1	qsys	Analyzes PH/PH/1 queues with phase-type arrivals and service distributions
randp	mam	Provides random value selection based on relative probability distributions
rap_sample	mam	Generates random samples from a Rational Arrival Process (RAP) marginal distribution
rl_env	rl	Provides a reinforcement learning environment interface for queueing networks
rl_env_general	rl	Provides a general reinforcement learning environment for queueing networks
rl_td_agent	rl	Implements a temporal difference learning agent for queueing network control
rl_td_agent_general	rl	Implements a general-purpose temporal difference learning agent for queueing
sn_deaggregate_chain_results	sn	Calculate class-based performance metrics for a queueing network based on performance measures of its chains
sn_fj_visits_spn	sn	Computes fork-join visit ratios by building auxiliary closed SPN models solved with SolverCTMC

Continued on next page

Table A.1 – API Function Reference. *Continued from previous page*

Function Name	Package	Description
sn_get_arv_r_from_tput	sn	Calculates the average arrival rates at each station from the network throughputs
sn_get_demands_chain	sn	Calculate new queueing network parameters after aggregating classes into chains
sn_get_node_arv_r_from_tput	sn	Computes per-node arrival rates from station throughputs by applying nodevisit ratios for non-station nodes
sn_get_node_tput_from_tput	sn	Computes per-node throughputs from station throughputs by combining nodevisit ratios with the routing matrix
sn_get_product_form_chain_params	sn	Calculate the parameters at class and chain level for a queueing network model
sn_get_product_form_params	sn	Extracts essential parameters (service demands, populations, visit ratios) from
sn_get_residt_from_respt	sn	Calculates the residence times at each station from the response times
sn_get_state_aggr	sn	Aggregates the state of the network
sn_has_class_switching	sn	Checks if the network uses class-switching
sn_has_closed_classes	sn	Checks if the network has one or more closed classes
sn_has_dps	sn	Check if the network includes a node with DPS scheduling
sn_has_dps_prio	sn	Check if the network includes a node with DPSPRIO scheduling
sn_has_fb	sn	Check if the network includes a node with FB (LAS) scheduling
sn_has_fcfs	sn	Identifies queueing networks using First-Come-First-Served scheduling disciplines
sn_has_fork_join	sn	Checks if the network uses fork and/or join nodes
sn_has_fractional_populations	sn	Checks if the network has closed classes with non-integer populations
sn_has_gps	sn	Check if the network includes a node with GPS scheduling
sn_has_gps_prio	sn	Check if the network includes a node with GPSPRIO scheduling
sn_has_hol	sn	Check if the network includes a node with HOL scheduling
sn_has_homogeneous_scheduling	sn	Checks if the network uses an identical scheduling strategy at every station
sn_has_inf	sn	Check if the network includes a node with INF scheduling
sn_has_joint_dependence	sn	Checks if the network has a station with joint-dependent service process (LJD or LJCD)
sn_has_lcfs	sn	Check if the network includes a node with LCFS scheduling
sn_has_lcfs_pr	sn	Check if the network includes a node with LCFSPR scheduling
sn_has_lcfs_pi	sn	Check if the network includes a node with LCFSPI scheduling
sn_has_lept	sn	Check if the network includes a node with LEPT scheduling
sn_has_ljf	sn	Check if the network includes a node with LJF scheduling
sn_has_lrpt	sn	Check if the network includes a node with LRPT scheduling
sn_has_load_dependence	sn	Checks if the network has a station with load-dependent service process
sn_has_mixed_classes	sn	Checks if the network has both open and closed classes
sn_has_multi_chain	sn	Check if the network has multiple chains
sn_has_multi_class	sn	Identifies queueing networks with multiple job classes, which require specialized
sn_has_multi_class_fcfs	sn	Checks if the network has any FCFS station with non-zero service rates for more than one class
sn_has_multi_class_heter_exp_fcfs	sn	Checks if the network has one or more stations with multiclass heterogeneous FCFS
sn_has_multi_class_heter_fcfs	sn	Checks if the network has one or more stations with multiclass heterogeneous FCFS
sn_has_multi_server	sn	Check if the network includes a multi-server node
sn_has_multiple_closed_classes	sn	Checks if the network has one or more closed classes
sn_has_open_classes	sn	Checks if the network has one or more open classes
sn_has_polling	sn	Check if the network includes a node with polling
sn_has_priorities	sn	Checks if the network uses class priorities
sn_has_product_form	sn	Determines if a queueing network has a known product-form solution by validating
sn_has_product_form_except_multi_class_heter_exp_fcfs	sn	Checks product-form assumptions except for multiclass heterogeneous exponential FCFS stations
sn_has_product_form_not_het_fcfs	sn	Checks if the network satisfies product-form assumptions (does not have heterogeneous FCFS)

Continued on next page

Table A.1 – API Function Reference. *Continued from previous page*

Function Name	Package	Description
sn_has_ps	sn	Check if the network includes a node with PS scheduling
sn_has_ps_prio	sn	Check if the network includes a node with PSPRIO scheduling
sn_has_psjf	sn	Check if the network includes a node with PSJF scheduling
sn_has_sd_routing	sn	Checks if the network has state-dependent routing strategies that violate product-form
sn_has_sept	sn	Check if the network includes a node with SEPT scheduling
sn_has_single_chain	sn	Check if the network has a single chain
sn_has_single_class	sn	Check if the network has a single class
sn_has_siro	sn	Check if the network includes a node with SIRO scheduling
sn_has_sjf	sn	Check if the network includes a node with SJF scheduling
sn_has_srpt	sn	Check if the network includes a node with SRPT scheduling
sn_is_closed_model	sn	Identifies closed queueing network models with finite job populations and no external
sn_is_mixed_model	sn	Checks if the network is a mixed model
sn_is_open_model	sn	Identifies open queueing network models with external arrivals and infinite
sn_is_population_model	sn	Checks if the model is a population model (only specific scheduling strategies without priorities or fork-join)
sn_is_state_valid	sn	Stochastic Network State Validation Utility
sn_modify_options	sn	Defines options (e.g., in-place vs. copy mode) controlling NetworkStruct modification methods
sn_nonmarkov_to_ph	sn	Converts non-Markovian distributions in a network to phase-type via Bernstein approximation
sn_print	sn	Prints comprehensive information about a NetworkStruct
sn_print_routing_matrix	sn	Prints the routing matrix of the network, optionally for a specific job class
sn_refresh_process_fields	sn	Refreshes service-process fields (mu, phi, proc, pie, phases) from current rates and SCVs
sn_refresh_visits	sn	Stochastic Network Visit Ratio Calculator
sn_rtnodes_to_rtorig	sn	Converts routing matrices from nodes to original format, specifically handling class switching nodes
sn_set_arrival	sn	Updates the arrival rate of an open class at a Source station; delegates to `sn_set_service`
sn_set_arrival_batch	sn	Batch update of arrival rates for multiple class indexes at a Source station
sn_set_fork_fanout	sn	Updates the fanout of a Fork node by writing to `nodeparam.fanOut`
sn_set_population	sn	Updates the population of a closed class and recalculates `nclosedjobs`
sn_set_priority	sn	Updates the scheduling priority of a class via the `classprio` vector
sn_set_routing	sn	Replaces the full routing matrix `rt` with optional auto-refresh of derived fields
sn_set_routing_prob	sn	Updates a single entry of `rt` for a (from,to) stateful node and class pair
sn_set_servers	sn	Updates the number of servers at a station via the `nservers` vector
sn_set_service	sn	Updates service rate and squared coefficient of variation for a (station,class) pair
sn_set_service_batch	sn	Batch update of service rates and SCV values for multiple (station,class) pairs; `NaN` entries are skipped
sn_set_service_coc	sn	Updates service rate in cell-of-cells representation, refreshing `mu`, `phi`, `pie`, and `proc`
sn_to_ag	sn	Converts a LINE network structure into the agent (RCAT) format used by SolverAG
sn_validate	sn	Validates NetworkStruct consistency and returns a list of validation errors
trace_mean	trace	Computes the arithmetic mean of empirical trace data. Fundamental statistical
trace_skew	trace	Computes the skewness of the trace data using Apache Commons Math
trace_var	trace	Computes sample variance and related statistics for empirical trace data
wf_analyzer	wf	Provides comprehensive workflow analysis capabilities including pattern
wf_auto_integration	wf	Provides automatic integration capabilities for workflow analysis with

Continued on next page

Table A.1 – API Function Reference. *Continued from previous page*

Function Name	Package	Description
wf_branch_detector	wf	Implements algorithms for detecting branching patterns in workflow traces
wf_loop_detector	wf	Implements algorithms for detecting loop and iterative patterns in workflow
wf_parallel_detector	wf	Implements algorithms for detecting parallel execution patterns in workflow
wf_pattern_updater	wf	Implements dynamic pattern updating algorithms for workflow analysis
wf_sequence_detector	wf	Implements algorithms for detecting sequential patterns in workflow traces

Bibliography

[1] M. Ajmone Marsan, G. Conte, and G. Balbo. A class of generalized stochastic petri nets for the performance evaluation of multiprocessor systems. *ACM Trans. Comput. Syst.*, 2(2):93–122, May 1984.

[2] Ian F Akyildiz and Gunter Bolch. Approximate analysis of load dependent general queueing networks. *IEEE Transactions on Software Engineering*, 14(11):1537–1545, 1988.

[3] D. Anick, D. Mitra, and M. M. Sondhi. Stochastic theory of a data handling system with multiple sources. *The Bell System Technical Journal*, 61:1871–1894, 1982.

[4] Simonetta Balsamo, Gian-Luca Dei Rossi, and Andrea Marin. A numerical algorithm for the solution of product-form models with infinite state spaces. In *European Performance Engineering Workshop*, volume 6342 of *LNCS*, pages 191–206. Springer, 2010.

[5] Y. Bard. Some extensions to multiclass queueing network analysis. In *Proc. of the 3rd Int'l Symp. on Model. and Performance Evaluation of Comp. Syst.*, pages 51–62, 1979.

[6] F. Baskett, K. M. Chandy, R. R. Muntz, and F. G. Palacios. Open, closed and mixed networks of queues with different classes of customers. *J. Assoc. Comput. Mach.*, 22:248–260, 1975.

[7] N. G. Bean, M-M. O'Reilly, and P. G. Taylor. Algorithms for return probabilities for stochastic fluid flows. *Stochastic Models*, 21:149–184, 2005.

[8] M. Bertoli, G. Casale, and G. Serazzi. The JMT simulator for performance evaluation of non-product-form queueing networks. In *Proc. of the 40th Annual Simulation Symposium (ANSS)*, pages 3–10, 2007.

[9] A. L. Bertozzi and J. McKenna. Multidimensional residues, generating functions, and their application to queueing networks. *SIAM Review*, 35(2):239–268, 1993.

[10] D. Bini, B. Meini, S. Steffé, J. F. Pérez, and B. Van Houdt. Smcsolver and q-mam: tools for matrix-analytic methods. *SIGMETRICS Performance Evaluation Review*, 39(4):46, 2012.

[11] A. Bobbio, A. Horváth, M. Scarpa, and M Telek. Acyclic discrete phase type distributions: properties and a parameter estimation algorithm. *Perform. Eval.*, 54(1):1–32, 2003.

[12] G. Bolch, S. Greiner, H. de Meer, and K. S. Trivedi. *Queueing Networks and Markov Chains*. Wiley, 2006.

[13] A. B. Bondi and W. Whitt. The influence of service-time variability in a closed network of queues. *Perform. Eval.*, 6:219–234, 1986.

[14] L. Bortolussi and M. Tribastone. Fluid limits of queueing networks with batches. In *Proceedings of the third joint WOSP/SIPEW international conference on Performance Engineering*, ICPE '12, pages 45–56, New York, NY, USA, 2012. ACM.

[15] L. Bright and P. G. Taylor. Calculating the equilibrium distribution in level dependent quasi-birth-and-death processes. *Stochastic Models*, 11(3):497–525, 1995.

[16] S. C. Bruell, G. Balbo, and P. V. Afshari. Mean value analysis of mixed, multiple class BCMP networks with load dependent service stations. *Performance Evaluation*, 4:241–260, 1984.

[17] Peter Buchholz, Jan Kriege, and Iryna Felko. *Input Modeling with Phase-Type Distributions and Markov Models: Theory and Applications*. Springer, Cham, 2014.

[18] J. P. Buzen. Computational algorithms for closed queueing networks with exponential servers. *Communications of the ACM*, 16(9):527–531, 1973.

[19] G. Casale. CoMoM: Efficient class-oriented evaluation of multiclass performance models. *IEEE Trans. Software Engineering*, 35(2):162–177, 2009.

[20] G. Casale. Exact analysis of performance models by the method of moments. *Perform. Eval.*, 68(6):487–506, 2011.

[21] G. Casale. Accelerating performance inference over closed systems by asymptotic methods. In *Proc. of ACM SIGMETRICS*. ACM Press, 2017.

[22] G. Casale. Integrated Performance Evaluation of Extended Queueing Network Models with Line. In *2020 Winter Simulation Conference (WSC)*, pages 2377–2388. IEEE, dec 2020.

[23] G. Casale, V. De Nitto Personé, and E. Smirni. QRF: An optimization-based framework for evaluating complex stochastic networks. *ACM Transactions on Modeling and Computer Simulation*, 26(3):15:1–15:24, 2016.

[24] G. Casale, P.G. Harrison, and O.W. Hong. Facilitating load-dependent queueing analysis through factorization. *Perform. Eval.*, 2021.

[25] G. Casale, Richard R. Muntz, and Giuseppe Serazzi. Geometric bounds: A noniterative analysis technique for closed queueing networks. *IEEE Trans. Computers*, 57(6):780–794, 2008.

[26] G. Casale, J. F. Pérez, and W. Wang. QD-AMVA: Evaluating systems with queue-dependent service requirements. In *Proceedings of IFIP PERFORMANCE*, 2015.

[27] G. Casale and M. Tribastone. Fluid analysis of queueing in two-stage random environments. In *Proceedings of QEST 2011*, 2011.

[28] G. Casale, M. Tribastone, and P. G. Harrison. Blending randomness in closed queueing network models. *Perform. Eval.*, 82:15–38, 2014.

[29] G. Casale, E. Z. Zhang, and E. Smirni. KPC-Toolbox: Best recipes for automatic trace fitting using Markovian arrival processes. *Performance Evaluation*, 67(9):873–896, 2010.

[30] K. M. Chandy and D. Neuse. Linearizer: a heuristic algorithm for queueing network models of computing systems. *Commun. ACM*, 25(2):126–134, 1982.

[31] H. Che, Y. Tung, and Z. Wang. Hierarchical web caching systems: Modeling, design and experimental results. *IEEE Journal on Selected Areas in Communications*, 20(7):1305–1314, 2002.

[32] G. L. Choudhury, K. K. Leung, and W. Whitt. Calculating normalization constants of closed queuing networks by numerically inverting their generating functions. *J. ACM*, 42(5):935–970, 1995.

[33] W.-M. Chow. Approximations for large scale closed queueing networks. *Perform. Eval*, 3(1):1–12, 1983.

[34] A. E. Conway. *Fast Approximate Solution of Queueing Networks with Multi-Server Chain-Dependent FCFS Queues*, pages 385–396. Springer US, Boston, MA, 1989.

[35] A. E. Conway and N. D. Georganas. RECAL - A new efficient algorithm for the exact analysis of multiple-chain closed queueing networks. *J. ACM*, 33(4):768–791, 1986.

[36] P. J. Courtois. *Decomposability: queueing and computer system applications*. Academic Press, New York, 1977.

[37] P.J. Courtois. Decomposability, instabilities, and saturation in multiprogramming systems. *Commun. ACM*, 18(7):371–377, 1975.

[38] P. Cremonesi, P. J. Schweitzer, and G. Serazzi. A unifying framework for the approximate solution of closed multiclass queuing networks. *IEEE Trans. Computers*, 51:1423–1434, 2002.

[39] A. Dan and D. Towsley. An approximate analysis of the LRU and FIFO buffer replacement schemes. *ACM SIGMETRICS Performance Evaluation Review*, 18(1):143–152, 1990.

[40] E. de Souza e Silva and R. R. Muntz. A note on the computational cost of the linearizer algorithm for queueing networks. *IEEE Trans. Computers*, 39(6):840–842, 1990.

[41] R.-A. Dobre, Z. Niu, and G. Casale. Approximating fork-join systems via mixed model transformations. In *Companion of the 15th ACM/SPEC International Conference on Performance Engineering*, ICPE '24 Companion, page 273–280, New York, NY, USA, 2024. Association for Computing Machinery.

[42] D. L. Eager and J. N. Lipscomb. The AMVA priority approximation. *Perform. Eval.*, 8(3):173–193, 1988.

[43] A. J. Field and P. G. Harrison. An approximate compositional approach to the analysis of fluid queue networks. *Perform. Eval.*, 64(9-12):1137–1152, October 2007.

[44] G. Franks. *Performance Analysis of Distributed Server Systems*. PhD thesis, Carleton, 1996.

[45] G. Franks, T. Al-Omari, M. Woodside, O. Das, and S. Derisavi. Enhanced modeling and solution of layered queueing networks. *IEEE Trans. Software Engineering*, 35(2):148–161, 2009.

[46] G. Franks, P. Maly, M. Woodside, D. C. Petriu, Alex Hubbard, and Martin Mroz. *Layered Queueing Network Solver and Simulator User Manual*. Carleton University, January 2013.

[47] G. Franks and M. Woodside. Multiclass multiservers with deferred operations in layered queueing networks, with software system applications. In *Proceedings of the 12th IEEE MASCOTS*, 2004.

[48] Giulio Garbi, Emilio Incerto, and Mirco Tribastone. Learning queuing networks by recurrent neural networks. In *Proc. of ACM/SPEC ICPE*, pages 56–66. ACM, 2020.

[49] N. Gast and B. Van Houdt. Transient and steady-state regime of a family of list-based cache replacement algorithms. *Queueing Syst*, 83(3-4):293–328, 2016.

[50] M. A. Gibson and J. Bruck. Efficient exact stochastic simulation of chemical systems with many species and many channels. *The Journal of Physical Chemistry A*, 104(9):1876–1889, 2000.

[51] D. T. Gillespie. Exact stochastic simulation of coupled chemical reactions. *J. Phys. Chem.*, 81(25):2340–2361, 1977.

[52] W. J. Gordon and G. F. Newell. Closed queuing systems with exponential servers. *Operations Research*, 15(2):254–265, 1967.

[53] V. Grassi, R. Mirandola, and A. Sabetta. From design to analysis models: a kernel language for performance and reliability analysis of component-based systems. In *Proc. 5th International Workshop on Software and Performance (WOSP)*, 2004.

[54] V. Gupta and P. G. Harrison. Fluid level in a reservoir with an on-off source. *SIGMETRICS Perform. Eval. Rev.*, 36(2):128–130, August 2008.

[55] A. Harel, S. Namn, and J. Sturm. Simple bounds for closed queueing networks. *Queueing Systems*, 31(1-2):125–135, 1999.

[56] P. G. Harrison and T. T. Lee. A new recursive algorithm for computing generating functions in closed queueing networks. In *Proc. of IEEE MASCOTS Symposium*, pages 223–230. IEEE Press, 2004.

[57] P. Heidelberger and K. Trivedi. Queueing network models for parallel processing with asynchronous tasks. *IEEE Trans. Computers*, 100(11):1099–1109, 1982.

[58] J. Hillston. *A Compositional Approach to Performance Modelling*. Cambridge University Press, 1996.

[59] G. Horton, V.G. Kulkarni, D.M. Nicol, and K.S. Trivedi. Fluid stochastic Petri nets: Theory, applications, and solution techniques. *European Journal of Operational research*, 105:184–201, 1998.

[60] G. Horváth. Measuring the distance between MAPs and some applications. In *Proc. of ASMTA 2015*, volume 9081 of *LNCS*, pages 95–109. Springer, 2015.

[61] G. Horváth and M. Telek. Sojourn times in fluid queues with independent and dependent input and output processes. *Performance Evaluation*, 79:160–181, 2014.

[62] G. Horváth and M. Telek. Butools 2: A rich toolbox for markovian performance evaluation. In *Proc. of VALUETOOLS*, pages 137–142, ICST, Brussels, Belgium, Belgium, 2017. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[63] J. R. Jackson. Jobshop-like queueing systems. *Management Science*, 10(1):131–142, 1963.

[64] A. Jensen. Markoff chains as an aid in the study of markoff processes. *Scandinavian Actuarial Journal*, 1953(sup1):87–91, 1953.

[65] F. P. Kelly. *Reversibility and Stochastic Networks*. Cambridge University Press, New York, NY, USA, 1979.

[66] F. P. Kelly. Loss networks. *Annals of Applied Probability*, 1(3):319–378, 1991.

[67] C. Knessl and C. Tier. Asymptotic expansions for large closed queueing networks with multiple job classes. *IEEE Trans. Computers*, 41(4):480–488, 1992.

[68] H. Kobayashi. Application of the diffusion approximation to queueing networks I: equilibrium queue distributions. *J. ACM*, 21(2):316–328, 1974.

[69] J. R. Koury, D. F. McAllister, and W. J. Stewart. Iterative methods for computing stationary distributions of nearly completely decomposable Markov chains. *SIAM Journal on Algebraic Discrete Methods*, 5(2):164–186, 1984.

[70] D.D. Kouvatsos. Entropy maximisation and queueing network models. *Annals of Operations Research*, 48:63–126, 1994.

[71] H. Koziolek. Performance evaluation of component-based software systems: a survey. *Perform. Eval.*, 67:634–658, 2010.

[72] W. Krämer and M. Langenbach-Belz. Approximate formulae for general single server systems with single and batch arrivals. *Angewandte informatik*, 9, 1978.

[73] V. G. Kulkarni. Fluid models for single buffer systems. In Jewgeni H. Dshalalow, editor, *Frontiers in queueing*, pages 321–338. CRC Press, Inc., Boca Raton, FL, USA, 1997.

[74] T. G. Kurtz. Solutions of ordinary differential equations as limits of pure jump Markov processes. *Journal of Applied Probability*, 7:49–58, 1970.

[75] T. G. Kurtz. *Approximation of Population Processes*. Society for Industrial and Applied Mathematics, 1981.

[76] G. Latouche and V. Ramaswami. *Introduction to Matrix Analytic Methods in Stochastic Modeling*. Society for Industrial and Applied Mathematics, 1999.

[77] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik. *Quantitative System Performance*. Prentice-Hall, 1984.

[78] P. L'Ecuyer, L. Meliani, and J. Vaucher. SSJ: A framework for stochastic simulation in Java. In *Proceedings of the 2002 Winter Simulation Conference*, pages 234–242. IEEE, 2002.

[79] C. Li, T. Altamimi, M. H. Zargari, G. Casale, and D. C. Petriu. Tulsa: A tool for transforming UML to layered queueing networks for performance analysis of data intensive applications. In *Quantitative Evaluation of Systems - 14th International Conference, QEST 2017, Berlin, Germany, September 5-7, 2017, Proceedings*, pages 295–299, 2017.

[80] Z. Li and G. Casale. Matrix network analyzer: A new decomposition algorithm for phase-type queueing networks. In *Proc. of ACM/SPEC ICPE*, pages 77–88, 2024.

[81] Z. Li and G. Casale. Matrix network analyzer: A new decomposition algorithm for phase-type queueing networks (work in progress paper). In *Companion of the 15th ACM/SPEC International Conference on Performance Engineering*, ICPE '24 Companion, page 34–39, New York, NY, USA, 2024. Association for Computing Machinery.

[82] J. D. C. Little. A proof for the queuing formula: $L = \lambda W$. *Operations Research*, 9(3):383–387, 1961.

[83] Andrea Marin and Samuele Rota Bulò. A general algorithm to compute the steady-state solution of product-form cooperating Markov chains. In *Proc. of MASCOTS 2009*, pages 515–524, 2009.

[84] Andrea Marin, Samuele Rota Bulò, and Simonetta Balsamo. A numerical algorithm for the decomposition of cooperating structured Markov processes. In *Proc. of the 20th IEEE MASCOTS*, pages 401–410. IEEE, 2012.

[85] KT Marshall. Some relationships between the distributions of waiting time, idle time and interoutput time in the gi/g/1 queue. *SIAM Journal on Applied Mathematics*, 16(2):324–327, 1968.

[86] J. McKenna and D. Mitra. Asymptotic expansions and integral representations of moments of queue lengths in closed markovian networks. *J. ACM*, 31(2):346–360, April 1984.

[87] Carl D. Meyer. Stochastic complementation, uncoupling markov chains, and the theory of nearly reducible systems. *SIAM Review*, 31(2):240–272, 1989.

[88] M. K. Molloy. Performance analysis using stochastic petri nets. *IEEE Trans. Computers*, 31(9):913–917, September 1982.

[89] Christoph Müller, Piotr Rygielski, Simon Spinner, and Samuel Kounev. Enabling fluid analysis for queueing petri nets via model transformation. *Electr. Notes Theor. Comput. Sci.*, 327:71–91, 2016.

[90] T. Murata. Petri nets: Properties, analisys and applications. *Proc. IEEE*, 77:541–579, 1989.

[91] M. F. Neuts. *Matrix-geometric solutions in stochastic models: an algorithmic approach*. Johns Hopkins University Press, 1981.

[92] M. F. Neuts. *Structured Stochastic Matrices of M/G/1 Type and Their Applications*. Marcel Dekker, 1989.

[93] M. Nuyens and A. Wierman. The foreground-background queue: A survey. *Performance Evaluation*, 65(3-4):286–307, 2008.

[94] J. F. Pérez and G. Casale. Assessing SLA compliance from Palladio component models. In *Proceedings of the 2nd MICAS*, 2013.

[95] J. F. Pérez and G. Casale. Line: Evaluating software applications in unreliable environments. *IEEE Trans. Reliability*, 66(3):837–853, Sept 2017.

[96] Juan F. Pérez, Giuliano Casale, and Sergio Pacheco-Sanchez. Estimating computational requirements in multi-threaded applications. *IEEE Trans. Software Eng.*, 41(3):264–278, 2015.

[97] Tuan Phung-Duc, Hiroyuki Masuyama, Shoji Kasahara, and Yutaka Takahashi. A simple algorithm for the rate matrices of level-dependent qbd processes. In *Proc. of QTNA*, pages 46–52. ACM, 2010.

[98] M. Reiser. Mean-value analysis and convolution method for queue-dependent servers in closed queueing networks. *Perform. Eval.*, 1:7–18, 1981.

[99] M. Reiser and S. Lavenberg. Mean-value analysis of closed multichain queuing networks. *J. ACM*, 27:313–322, 1980.

[100] A. Riska and E. Smirni. MAMSolver: A matrix analytic methods tool. In *Proc. of the 12th Int. Conf. on Modelling Techniques and Tools (TOOLS)*, volume 2324 of *LNCS*, pages 205–211, 2002.

[101] A. Riska and E. Smirni. ETAQA solutions for infinite Markov processes with repetitive structure. *INFORMS Journal on Computing*, 19(2):215–228, 2007.

[102] J. A. Rolia and K. C. Sevcik. The method of layers. *IEEE Trans. Software Engineering*, 21(8):689–700, August 1995.

[103] J. Ruuskanen, T. Berner, K.-E. Årzén, and A. Cervin. Improving the mean-field fluid model of processor sharing queueing networks for dynamic performance models in cloud computing. *Perform. Evaluation*, 151:102231, 2021.

[104] R. Sahner, K. S. Trivedi, and A. Puliafito. *Performance and Reliability Analysis of Computer Systems: An Example-Based Approach Using the SHARPE Software Package*. Kluwer Academic Publishers, 1996.

[105] W. Scheinhardt. *Markov-modulated and feedback fuid queues*. PhD thesis, University of Twente, 1998.

[106] L. Schrage. A proof of the optimality of the shortest remaining processing time discipline. *Operations Research*, 16(3):687–690, 1968.

[107] P. J. Schweitzer. Approximate analysis of multiclass closed networks of queues. In *Proc. of the Int'l Conf. on Stoch. Control and Optim.*, pages 25–29, Amsterdam, 1979.

[108] K. Sevcik. Priority scheduling disciplines in queuing network models of computer systems. In *IFIP Congress*, 1977.

[109] M. Sheldon and G. Casale. Taussa: Simulating markovian queueing networks with tau leaping. *SIGMETRICS Perform. Eval. Rev.*, 49(4):70–75, jun 2022.

[110] Simon Spinner, Giuliano Casale, Fabian Brosig, and Samuel Kounev. Evaluating approaches to resource demand estimation. *Performance Evaluation*, 92:51–71, 2015.

[111] W. J. Stewart. *Introduction to the Numerical Solution of Markov Chains*. Princeton University Press, 1994.

[112] R. Suri, S. K. Sahu, and M. Vernon. Approximate mean value analysis for closed queuing networks with multiple-server stations. In *Proc. of the Industrial Engineering Research Conference*, pages 1–6, 2007.

[113] H. Takagi. Queuing analysis of polling models. *ACM Computing Surveys*, 20(1):5–28, 1988.

[114] Y. Takahashi. A lumping method for numerical calculations of stationary distributions of Markov chains. Technical Report B-18, Department of Information Sciences, Tokyo Institute of Technology, Tokyo, Japan, June 1975.

[115] M. Tribastone. A fluid model for layered queueing networks. *IEEE Trans. Software Engineering*, 39(6):744–756, June 2013.

[116] M. Tribastone, J. Ding, S. Gilmore, and J. Hillston. Fluid rewards for a stochastic process algebra. *IEEE Trans. Software Engineering*, 38(4):861–874, 2012.

[117] H. Wang and K. C. Sevcik. Experiments with improved approximate mean value analysis algorithms. *Perform. Eval.*, 39(1-4):189–206, 2000.

[118] W. Wang, G. Casale, and C. A. Sutton. A bayesian approach to parameter inference in queueing networks. *ACM Trans. Model. Comput. Simul.*, 27(1):2:1–2:26, 2016.

[119] Weikun Wang, Giuliano Casale, Ajay Kattepur, and Manoj K. Nambiar. Maximum likelihood estimation of closed queueing network demands from queue length data. In *Proc. of ACM/SPEC ICPE*, pages 3–14. ACM, 2016.

[120] W. Whitt. The queueing network analyzer. *Bell System Technical Journal*, 62(9):2779–2815, 1983.

[121] A. Wierman and M. Harchol-Balter. Classifying scheduling policies with respect to unfairness in an M/GI/1. In *Proc. ACM SIGMETRICS*, pages 238–249, 2003.

[122] J. Zahorjan, D. L. Eager, and H. M. Sweillam. Accuracy, speed, and convergence of approximate mean value analysis. *Perform. Eval.*, 8(4):255–270, 1988.

[123] S. Zhou and M. Woodside. A multiserver approximation for cloud scaling analysis. In *Companion of the 2022 ACM/SPEC International Conference on Performance Engineering*, ICPE '22, page 129–136, New York, NY, USA, 2022. Association for Computing Machinery.